Trabajo Fin de Máster

Máster en Ingeniería Industrial

KdUINO PRO:
Módulo de integración de sensores de bajo coste y bajo consumo para entorno acuático

MEMORIA

Autor: Diego Iraburu Zulaica
Director: Vicenç Parisi Baradad

Convocatòria: 2019

Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona

Resumen

Este Trabajo Final de Máster consistirá en el diseño y desarrollo de un **módulo de integración de sensores de bajo coste y bajo consumo para instrumentación del entorno marino**, llamado **KdUINO Pro.** Los parámetros que medirá este módulo son el **coeficiente de atenuación difusa del agua Kd y la temperatura**, aunque se podrían implementar otro tipo de medidas en un futuro. El módulo principal, el microcontrolador maestro, se podrá comunicar con las demás unidades independientes derivadas de él vía UART. Las unidades independientes están constituídas por un microcontrolador, un sensor óptico de luz y otro de temperatura. El microcontrolador maestro se ubicaría en una boya en la superficie y las unidades independientes en el interior de un tubo a diferentes profundidades.

El objetivo del proyecto, KdUINO Pro, es encontrar una alternativa de bajo coste y bajo consumo a los sensores acuáticos actuales. Se logra diseñar el módulo con un presupuesto mínimo de 180,34€, mientras que módulos más sofisticados se encuentran en torno a 1000€. Otro de los objetivos es que el proyecto se incluya en el movimiento **DIY** (Do It Yourself), de tal manera que cualquier usuario tenga disponible la información del diseño del módulo y pueda realizarlo por su cuenta, con sus propios medios. Con esta última consideración, se pretende que los usuarios que realicen su propio KdUINO Pro puedan enviar datos recogidos a estaciones que requieran esa información para obtener un muestreo más completo.

Este proyecto solamente valida la **parte electrónica de software y hardware** del KdUINO Pro, no diseña la cubierta que le permitiría sumergirse en el agua. Además, al final de la memoria se muestran opciones de mejora del módulo para una implementación futura.

ETSEIB

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. Glosario

Coeficiente de atenuación difusa Kd: es un parámetro que estima la claridad o turbidez del agua midiendo la capacidad de penetración de la radiación solar incidente en ella.

Coeficiente de determinación R^2: estadístico que determina la calidad del modelo para replicar los resultados, y la proporción de variación de los resultados que puede explicarse por el modelo

DIY (Do It Yourself): práctica de la fabricación o reparación de cosas por uno mismo, de modo que se ahorra dinero, se entretiene y se aprende al mismo tiempo.

IoT (Internet Of Things): concepto que se refiere a una interconexión digital de objetos cotidianos con internet.

LoPy4: Microcontrolador de la marca Pycom utilizado en este proyecto.

Microcontrolador principal / maestro: Microcontrolador que gestiona los datos transmitidos por parte de los esclavos y se encarga de almacenar y hacer recibir al usuario los datos deseados.

Microcontrolador secundario / esclavo: Microcontrolador que se encarga de obtener los datos de los sensores y enviarlos al maestro.

Open-source: programa informático que ofrece a cualquier usuario el acceso a su código de programación la modificación del mismo.

UART (Universal Asynchronous Receiver-Transmitter): dispositivo que controla los puertos y dispositivos serie. Se encuentra integrado en la placa base o en la tarjeta adaptadora del dispositivo.

2. Prefacio

La instrumentación marina, y todo lo relacionado con la oceanografía, es precisa y sofisticada, pero, en general, tiene un alto coste. Este inconveniente, tiene como consecuencia que el presupuesto se invierta generalmente en un número escaso de instrumentos electrónicos. De esta manera, los resultados obtenidos son poco fiables y de difícil interpretación debido al limitado número de puntos de muestreo conseguido.

KdUINO Pro es una solución alternativa para realizar medidas del medio acuático con un coste y consumo menor, permitiendo, con una misma inversión, tener un mayor número de instrumentos de medida para realizar una cantidad mayor de muestreos. Al permitir un mayor número de muestreos, los resultados que se obtendrán serán más fiables y se ajustarán más a la realidad.

El coste de realizar el prototipo será de un orden de 5 veces inferior a las soluciones actuales y el diseño será fácilmente accesible y realizable, por lo que cualquier usuario interesado en ello podrá aportar más muestreos con su propio KdUINO Pro.

2.1. Origen del proyecto

El origen de este proyecto se debe a un proyecto anterior, llamado **KdUINO** *(Bardají, Sánchez, Simon, Wernand, & Piera, 2016),* que se elaboró como un primer prototipo del módulo. La idea era realizar un módulo de bajo consumo y bajo coste, más simple que el KdUINO Pro. Un único microcontrolador maestro conectaba varios sensores ópticos de luz y se colocaban a diferentes profundidades.

El prototipo fue capaz de obtener el parámetro Kd, almacenarlo y enviarlo, así como enviar datos de GPS. Sus mayores inconvenientes eran el número limitado de sensores, la distancia entre instrumentos comunicándose (debido a que se utilizaba I_2C), que estaba adaptado a un número predeterminado de sensores y la poca flexibilidad que ofrecía, entre otros.

Este proyecto, KdUINO Pro, pretende optimizar el KdUINO en cuanto a nivel de electrónica.

La oportunidad de colaborar en el diseño de este módulo ha sido gracias al CSIC (Consejo Superior de Investigaciones Científicas) y a la colaboración entre el ICM (Instituto de Ciencias del Mar) y la UPC (Universidad Politécnica de Cataluña).

2.2. Motivación

El agua abarca, aproximadamente, un 70% de la superficie del planeta. Se trata de un campo de estudio muy extenso en el que recoger información es muy complicado. Por ello, se suele realizar un muestreo sobre la superficie para segmentar y limitar las medidas a tomar.

Los sensores de alto coste permiten realizar un estudio muy preciso, con mucha fiabilidad. Sin embargo, debido al gran coste de la instrumentación oceanográfica y su operación, la resolución espacial se resiente y los estudios de grandes zonas marinas se realizan midiendo en pocos puntos e interpolando datos para intentar mitigar esas zonas de estudio donde no se han podido obtener datos reales.

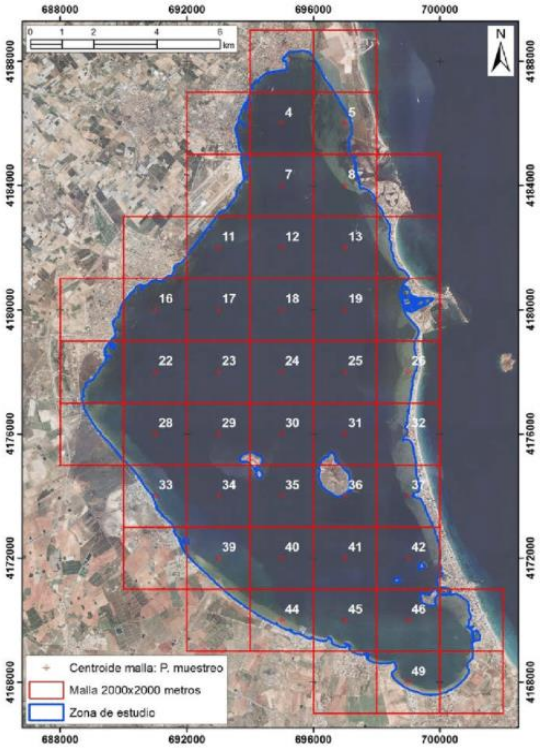


Figura 1: Mapa de muestreo del Mar Menor sectorizado (Espla, 2019)

Este tipo de estrategia, a la hora de realizar el estudio de una porción de la superficie del mar, puede llevar a errores de interpretación del comportamiento del medio.

En cambio, si se obtuviesen un mayor número de muestras, el comportamiento estimado sería más cercano al real. La *Figura 2* muestra la diferencia entre una interpretación de un muestreo con muchas muestras o con pocas. En esta imagen se ve que a pesar de que las medidas (puntos azules) no son muy precisas, se puede estimar un patrón (línea azul) que explica correctamente la tendencia de un evento (línea negra). Los puntos rojos son los puntos medidos con menos instrumentos pero más precisos y la línea roja es la interpolación creada con estos puntos.

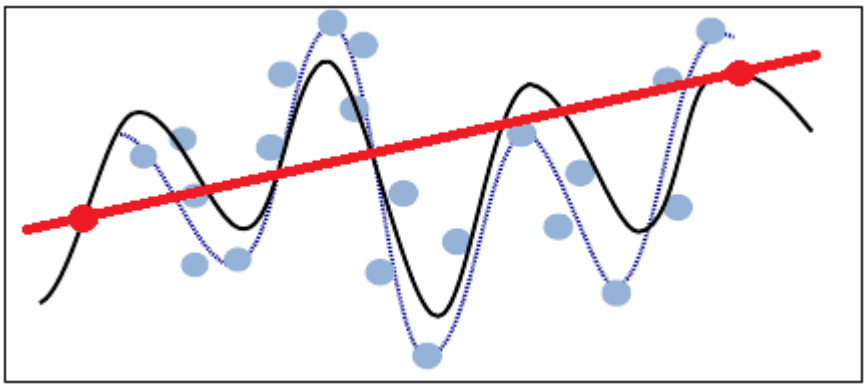


Figura 2: Interpretación con más cantidad de muestras (azul) y con menos (rojos) (Córdoba, 2011).

Se busca conseguir un módulo de sensores que disminuya su coste total para poder, con la misma inversión, obtener más muestras. La desventaja del KdUINO Pro frente a otros instrumentos de medida más sofisticados será que las muestras tomadas se desviarán más del valor real. Sin embargo, la magnitud de este desvío seguirá estando dentro de unos límites aceptables.

3. Introducción

Este proyecto consiste en validar la parte de software y hardware del KdUINO Pro. Una vez validado y depurado el diseño electrónico, se trabajará en un futuro el diseño mecánico del KdUINO Pro implementando elementos que le permitan sumergirse en el agua sin dañar la parte electrónica validada. Esta última parte no se realizará por escasez de tiempo y recursos.

Cabe recordar que el actual proyecto se basa en un prototipo ya testeado, el KdUINO, mencionado en el apartado *2. Prefacio*.

3.1. Objetivos del proyecto

Los objetivos que se han establecido en este proyecto son los siguientes:

- Encontrar una **alternativa de bajo coste y consumo** frente a la tecnología actual de alto coste.

- Conseguir que el proyecto sea simple y accesible (**open-source**) para cualquier persona interesada en él. De esta manera, el proyecto entraría dentro del movimiento **DIY** (Do It Yourself).

- Sustituir los sensores conectados al maestro del KdUINO, por la implementación de **unidades independientes** que lleven un **microcontrolador, un sensor de luz y otro de temperatura** en el KdUINO Pro. Permitir la **implementación de cualquier cantidad de estas unidades esclavo** sin que el total afecte al correcto comportamiento del módulo.

- Estudiar e implementar el **método de transmisión de datos óptimo entre microcontroladores**, ya sea por comunicación inalámbrica o con cables.

- Elegir el protocolo de comunicación óptimo para la **transmisión y recepción de datos desde el microcontrolador principal al usuario**.

- **Diseñar y validar** la parte de software y hardware del KdUINO Pro. Esto implica dotar al KdUINO Pro de funciones útiles a nivel de sofware y un diseño sólido y fiable a nivel de hardware.

3.2. Alcance del proyecto

El alcance de este proyecto se concreta en los siguientes puntos:

- Implementación del código que permita la **transmisión y recepción** de datos recogidos por los sensores de los **microcontroladores esclavos con el microcontrolador maestro**. Permitir que el número de esclavos no afecte al comportamiento del código.

- Realizar la programación que permita la **transmisión de datos del microcontrolador maestro a un receptor**, mediante comunicación inalámbrica.

- Implementación del código que permite recoger las **medidas de cada sensor óptico de luz y de temperatura**.

- Realizar la programación para almacenar los datos recogidos en una **tarjeta SD,** así como la obtención de los datos **GPS** y **nivel de batería**.

- Realizar la programación para calcular el **parámetro Kd y el coeficiente de determinación R^2** a partir de las medidas de luz.

- Conseguir que el módulo entero entre en un modo **Deep-sleep** permitiendo ahorrar batería.

- Realización de las **conexiones** eléctricas del microcontrolador principal y los demás microcontroladores.

- **Testeado y optimización** del correcto comportamiento de la recepción, comunicación y almacenamiento de datos.

- Proponer **alternativas y mejoras** en varios niveles del proyecto.

4. Estudio del arte

Existen varios métodos para medir la transparencia del agua. El método más antiguo y elemental es el **disco de Secchi**. Se trata de un sencillo instrumento para comprobar la transparencia del agua de lagos y océanos. Está fabricado en latón lacado y tiene un diámetro de 200 mm. El disco de Secchi se introduce en el agua y obtiene la lectura de la profundidad por las marcas del cabo. Cuando el disco ya no se puede ver, se suelta 0,5m más y luego se sube lentamente. La segunda lectura se hace cuando el disco se puede discernir de nuevo. Es en este punto cuando hay que calcular aritméticamente, de las dos lecturas, la visibilidad de la profundidad (Consultores, 2014).

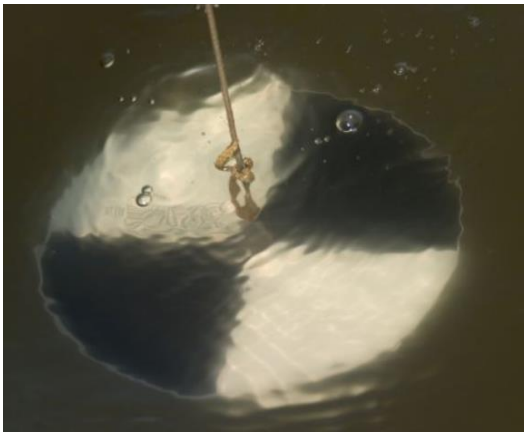


Figura 3: Disco de Secchi (Consultores, 2014)

El método del **tubo de turbidez** es similar, pero insertando el disco en un tubo para que el movimiento del agua no altere su posición.

Por otro lado, existe tecnología capaz de conseguir datos del agua de forma automática y precisa que facilita enormemente la recogida de información en varios puntos de muestreo. Se comenzará comentando la **tecnología de alto coste**.

Existe por ejemplo el sensor *RAMSES* de *TriOS*, un sensor que permite medir la irradiancia de un modo muy preciso en cualquier longitud de onda deseada del espectro de luz. Este sensor tiene un elevado coste y para monitorizar los datos se necesita otro instrumento. Lo mismo sucede con el sensor *LI-192 Underwater Quantum Sensor* de la marca *LI-COR*.

Figura 4: De izquierda a derecha, sensor RAMSES, sensor LI-192 Underwater Quantum y el módulo PRR-800 *(RSHydro, 2019)*, *(Frondriest, 2019)*, *(BiosphericalInstrumentsInc, 2011)*.

Existe un módulo ya totalmente equipado que es el *PRR-800,* pero su precio es incluso más elevado.

Ante la necesidad de conseguir una tecnología de medida bajo el agua a un precio inferior, han ido surgiendo proyectos de iniciativa **open source** que lograron ser implementados y conseguir las medidas deseadas de una forma fiable. Todos los proyectos citados a continuación han sido mencionados en el estudio del arte previo realizado para el proyecto KdUINO (Bardají & Piera, 2013).

A continuación se citaran los proyectos más interesantes en la actualidad:

- ***Spectruino***: Sensor espectómetro de Arduino para medir la intensidad de la luz en un gran rango de longitudes de onda. Su precio es de 200-250€.

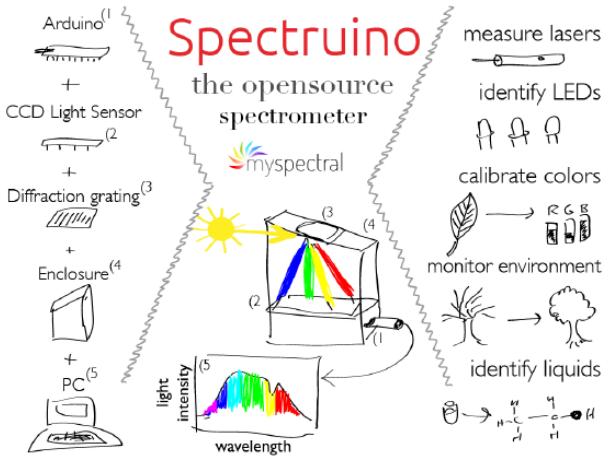


Figura 5: Spectruino (Bardají & Piera, 2013)

- ***Diveduino***: Proyecto basado en la plataforma Arduino que permite almacenar una tarjeta SD los datos de profundidad y temperatura y también mostrarlos en una pantalla.

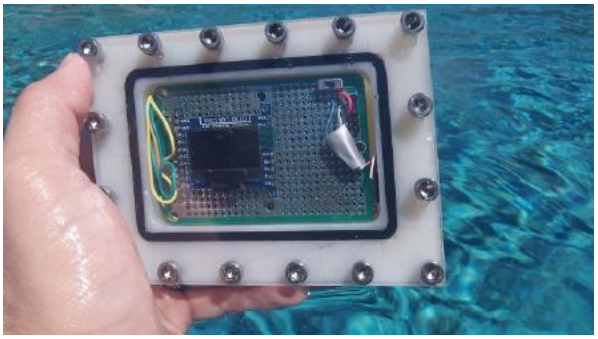


Figura 6: Diveduino (Bardají & Piera, 2013)

- ***Coconut Pi*:** Proyecto creado por unos estudiantes de la universidad de Singapur que consiste en un robot submarino que usa Raspberry Pi para almacenar y Arduino para controlar los movimientos del robot. Está hecho con un tupperware que se asemeja a la idea de la boya de KdUINO Pro.

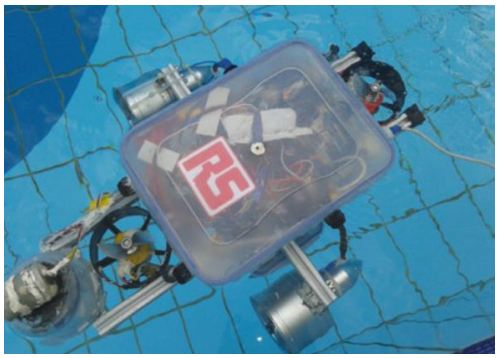


Figura 7: Coconut Pi (Bardají & Piera, 2013)

Como se puede apreciar, existen los medios para crear proyectos interesantes con tecnología de menor coste y puede suponer un ahorro de inversión de una gran magnitud. Además, se opta por utilizar plataformas de código abierto para poner al alcance de todos estos diseños.

El KdUINO Pro se encuentra en este último campo de tecnología mencionado. Utilizará una plataforma open-source, elementos electrónicos de bajo coste y al alcance de todos, diseños mecánicos usando utensilios cotidianos y/o accesibles y una idea simple y sencilla para obtener los objetivos deseados.

5. Cálculo del coeficiente de atenuación difusa Kd

Antes de explicar en detalle el diseño del KdUINO Pro, se explicará cómo se calcula el parámetro **Kd, coeficiente de atenuación difusa**, que determina la turbidez del agua. El KdUINO Pro también registrará las temperaturas a diferentes profundidades, pero su obtención no requiere cálculos, el sensor obtendrá el valor directamente.

La ley de Beer-Lambert relaciona la absorción de la luz con las propiedades del medio por donde la luz viaja. En particular, cuando el medio es líquido, la intensidad de la luz disminuye exponencialmente como una función de la profundidad descrita de la siguiente manera:

$$E_2 = E_1 e^{-k_d \Delta z}$$

Ecuación 1: Beer-Lambert en medio líquido

Donde E_2 es la irradiancia de la luz a la profundidad z_2, E_1 es la irradiancia de la luz a la profundidad z_1, y Δz es la diferencia entre z_2 - z_1 en metros. E_2 y E_1 son dependientes de la longitud de onda, pero se omitirá esta dependencia por simplicidad. De esta manera, el coeficiente Kd se obtiene a partir de la pendiente de la recta que se obtiene aplicando el logaritmo en la *Ecuación 1*:

$$ln\,\mathrm{E}_2 = -\mathrm{K}_\mathrm{d}\Delta\mathrm{z} + ln(\mathrm{E}_1)$$

Ecuación 2: Logaritmo de la ecuación de Beer-Lambert en medio líquido

Si E_1 es la irradiancia de la luz aproximadamente a la altura de la superficie del medio líquido *(z1 = 0),* es posible extraer la irradiancia de la luz a cualquier profundidad z_i en la columna de agua de la siguiente manera:

$$\ln E_i = -K_d \Delta z + \mathrm{constant}$$

Ecuación 3: Logaritmo de la irradiancia de luz a cualquier profundidad

Como se muestra en la siguiente figura, usando la regresión linear de un conjunto de medidas de luz a diferentes profundidades, Kd se consigue fácilmente como el negativo de la pendiente de la regresión linear *(Bardají, Sánchez, Simon, Wernand, & Piera, 2016)*.

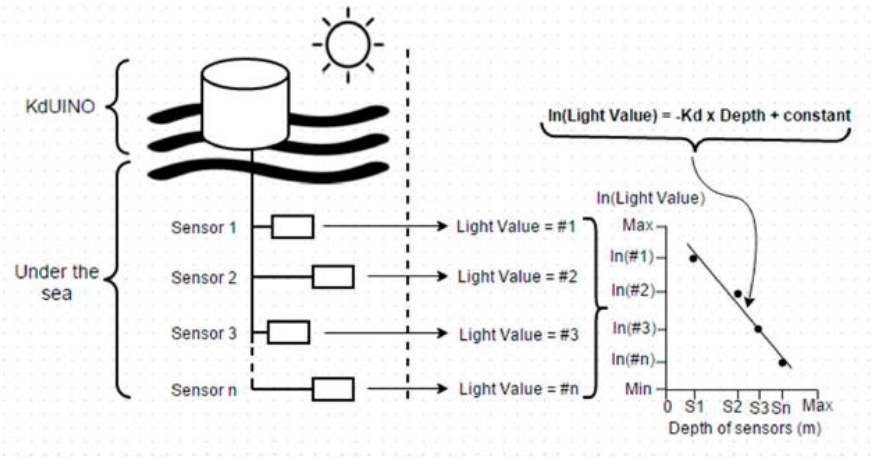


Figura 8: Obtención del coeficiente de atenuación difusa Kd (Bardají, Sánchez, Simon, Wernand, & Piera, 2016)

Las medidas tomadas no coinciden exactamente con la recta aproximada calculada. Por ello, será necesario también calcular el **coeficiente de determinación de la regresión linear R^2** para indicar el grado de fiabilidad del cálculo del coeficiente de atenuación difusa Kd.

Cuanto más cercano al valor 0 sea el parámetro Kd, mayor grado de transparencia tendrá el volumen de agua muestreada. Mientras que cuanto más se aleje de este valor mayor grado de turbidez tendrá el agua tratada.

Por otra parte, el coeficiente de determinación de la regresión linear, cuanto más se acerque a 1, más exacta es la relación lineal, y cuanto más se aproxime a 0, significa que la relación lineal es mínima.

6. Electrónica del módulo

En este apartado, se describirá brevemente cada elemento del módulo elegido y, en algunos casos, otras opciones compatibles con el diseño del KdUINO Pro.

Varios de los elementos utilizados han sido elegidos debido a que el Instituto de Ciencias del Mar ya disponía de ellos y se habían realizado otros trabajos con esta tecnología.

6.1. Sensor óptico de luz

El sensor óptico de luz elegido para el proyecto ha sido el sensor ***TCS34725*** de la marca *Adafruit*.

Este sensor es capaz de medir en las longitudes de RGB y elementos de detección de luz clara. Un filtro de bloqueo de IR minimiza el componente espectral de IR de la luz entrante y permite que las mediciones de color se realicen con precisión. El sensor también tiene un gran rango dinámico de 3.800.000:1, con tiempo de integración y ganancia ajustables, por lo que es adecuado para usar detrás de un vidrio oscuro.

El TCS34725 permite conectarse a cualquier microcontrolador con **I^2C** y tiene bibliotecas de funciones software disponibles. Este sensor en concreto es un sensor digital, por lo que no hace falta realizar una conversión A/D para almacenar los datos, lo que disminuye el consumo eléctrico (Adafruit, s.f.).

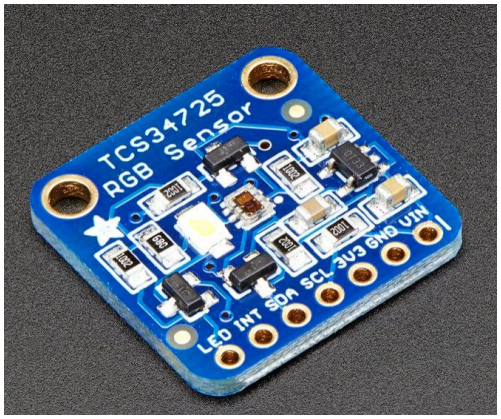


Figura 9: Sensor TCS34725 de Adafruit (Adafruit, s.f.)

Aunque se ha optado por utilizar este sensor en concreto (debido a la disponibilidad en el centro donde se realizó este proyecto), existe una gran cantidad de sensores ópticos de luz que podrían valer para realizar la adquisición de datos.

6.2. Sensor de temperatura

Si se quiere medir la temperatura del agua, este sensor debe ser impermeable. El Instituto de Ciencias del Mar ya disponía de los sensores que se usarán, *Waterproof **DS18B20** Digital temperature sensor* de la marca *Adafruit*.

Este sensor es útil para medir a una distancia lejana al microcontrolador, por su longitud, y en condiciones húmedas. Si bien el sensor funciona correctamente hasta 125 °C, el cable está revestido en PVC, por lo que se sugiere mantenerlo por debajo de 100°C. Debido a que es digital, no se produce ninguna degradación de la señal, incluso en largas distancias. Estos sensores de temperatura digitales de un cable son bastante precisos (±0.5°C en gran parte del rango) y puede utilizarse con sistemas de 3 a 5V.

El único inconveniente es que utilizan el protocolo *Dallas 1-Wire*, que es algo complejo y requiere gran cantidad de código para analizar la comunicación. Además, hay que conectar una resistencia pull-up de 4,7k, entre el pin y 5V (Adafruit, s.f.).

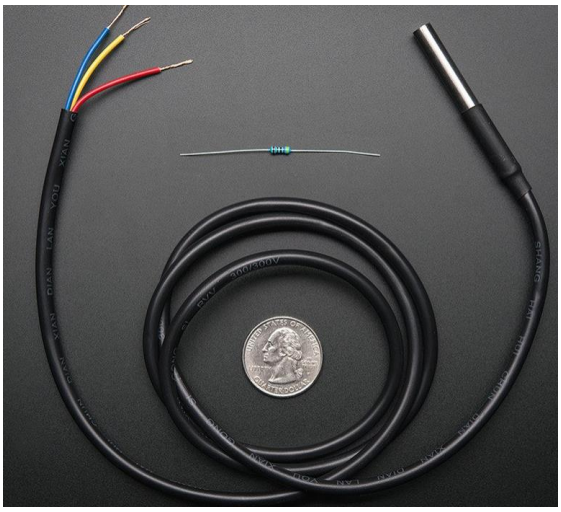


Figura 10: Waterproof DS18B20 Digital temperature sensor de Adafruit (Adafruit, s.f.)

Al igual que en el caso del sensor óptico, existe una gran cantidad de sensores de temperatura a prueba de agua en el mercado y a un coste muy similar. Se podría utilizar cualquiera teniendo en cuenta que el código variaría debido a la librería utilizada.

6.3. Microcontrolador

En el proyecto del KdUINO *(Bardají, Sánchez, Simon, Wernand, & Piera, 2016)*, se proponía utilizar Arduino (de ahí provenía su nombre) o un microcontrolador similar, como Teensy o BeagleBone entre otros. Arduino es un microcontrolador diseñado para hacer el proceso de añadir componentes electrónicos muy simple. Esta opción tiene el valor de que es barato y fácil de programar. Además, la comunidad de usuarios de Arduino es muy extensa.

Tabla 1: Opciones de Arduino

Nombre	Procesador	Frecuencia (MHz)	Hoja de especificaciones
Arduino Leonardo	Atmega32u4	16	http://arduino.cc/en/Main/ArduinoBoardLeonardo
Arduino Uno	ATmega328P	16	http://arduino.cc/en/Main/ArduinoBoardUno
Arduino Due	AT91SAM3X8E	84	http://arduino.cc/en/Main/ArduinoBoardDue
Arduino Mega2560	ATmega2560	16	http://arduino.cc/en/Main/ArduinoBoardDue
Arduino Ethernet	ATmega328	16	http://arduino.cc/en/Main/ArduinoBoardEthernet
Arduino Fio	ATmega328P	8	http://arduino.cc/en/Main/ArduinoBoardFio
Arduino Mini	ATmega168	16	http://arduino.cc/en/Main/ArduinoBoardProMini
Arduino Nano	ATmega328	16	http://arduino.cc/en/Main/ArduinoBoardNano

A pesar de que Arduino se trata de una buena opción, el **KdUINO Pro** opta por utilizar el microcontrolador **LoPy4 de Pycom** por su potencial y mayor número de funcionalidades. El LoPy4 es una placa de desarrollo microPython compacta para red cuádruple (LoRa, Sigfox, WiFi, Bluetooth). MicroPython es una implementación de Python 3.5 optimizado para ser ejecutado en microcontroladores. Permite un desarrollo más rápido y simple que usando C.

Esta plataforma de **IoT** (Internet Of Things) es perfecta para disponer de los instrumentos conectados. Con el último chipset *Espressif ESP32*, ofrece una buena combinación de potencia y flexibilidad (Pycom, s.f.).

Al mismo tiempo, el microcontrolador maestro estará conectado a una placa de expansión **Pytrack** de Pycom. Esta placa añade las funcionalidades de GPS, acelerómetro de 3 ejes, acceso a puerto USB, cargador de batería LiPo, compatibilidad con tarjetas MicroSD y una operación de DeepSleep (sueño profundo).

Figura 11: microcontrolador Lopy4 y placa Pytrack de Pycom (Pycom, s.f.).

Dentro de los productos de la marca Pycom, LoPy4 es la placa de menor coste, en la franja de los microcontroladores que ofrecen la comunicación LoRa y Sigfox. Los microcontroladores de menor precio no ofrecen las prestaciones del LoPy4, pero se podría modificar el proyecto para adaptarse a ellos en caso de preferir otro producto.

Tabla 2:Opciones de Pycom

Nombre	Procesador	Precio (€)	Hoja de especificaciones
WiPy 3.0	Espressif ESP32 chipset	19.95	https://pycom.io/product/wipy-3-0/
SiPy	Espressif ESP32 chipset	24.95	https://pycom.io/product/sipy/
Lopy4	Espressif ESP32 chipset	29.95	https://pycom.io/product/lopy4/
GPy	Espressif ESP32 SoC	44.00	https://pycom.io/product/gpy/
FiPy	Espressif ESP32 SoC	54.00	https://pycom.io/product/fipy/

Existen a su vez otras opciones para la placa de expansión con diferentes funcionalidades, se eligió Pytrack porque es la única que ofrece la función GPS y es necesaria para el proyecto.

6.4. Antenas

Al haber elegido el microcontrolador Lopy4, la marca Pycom ofrece el kit de antenas.

Figura 12: Kit de antenas de Pycom (Pycom, s.f.).

Hay muchos tipos de antenas en el mercado y sus costes asociados son muy diferentes. La elección depende de los valores deseados de ganancia, directividad, eficiencia, etc.

Tabla 3: Opciones de antenas

Producto	Frecuencia (GHz)	Ganancia (dB)	Precio (€)	Hoja de especificaciones
ANT-24G-905-SMA	2.4-2.5	5	7.56	http://docs-europe.electrocomponents.com/webdocs/0e0b/0900766b80e0bbd5
ANT-24G-HL90-SMA	2.4-2.5	0	7.41	http://docs-europe.electrocomponents.com/webdocs/0ba3/0900766b80ba38b8
ANT-24G-WHJ-SMA	2.4-2.5	0	8.34	http://docs-europe.electrocomponents.com/webdocs/0ba3/0900766b80ba38b8
ANT-24G-DPL-FP	2.4-2.5	2.1	11.49	http://docs-europe.electrocomponents.com/webdocs/0ba3/0900766b80ba38b8
ANT-SS2.4G	2.4	0	4.35	http://docs-europe.electrocomponents.com/webdocs/0df8/0900766b80df8ad9
FBKR35068-SM-KR	2.4	2	8.93	http://docs-europe.electrocomponents.com/webdocs/0ba1/0900766b80ba126a
ANT-2.4G	2.4	10	14.89	http://docs-europe.electrocomponents.com/webdocs/0793/0900766b80793024

6.5. Transceptor para protocolo RS485

Para usar el estándar **RS485** con el microcontrolador, se ha decidido utilizar el transceptor MAX485 de la marca MaximIntegrated. El MAX485 es un transceptor de baja potencia para comunicaciones RS-485 y RS422 half-duplex y puede transmitir hasta 2.5Mbps. Consta de 8 pines que permiten el envío y recepción de datos (RO y DI), su habilitación (DE y RE), las líneas de comunicación del protocolo RS485 (A y B), Vcc y tierra (MaximIntegrated, s.f.).

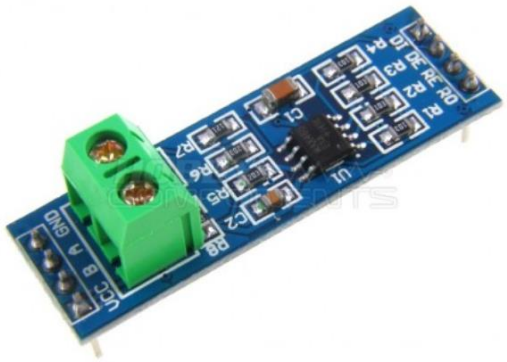


Figura 13: Transceptor MAX485 (MaximIntegrated, s.f.)

6.6. Cubierta protectora

El elemento descrito en este apartado no se llega a implementar en el proyecto. Se menciona para dar una idea que permita implementar también la parte mecánica del KdUINO Pro al usuario que lea este proyecto.

La cubierta que protege la electrónica del módulo debe ser totalmente estanca, que impida la entrada de agua. Para ello se utilizaría un tubo de una longitud y diámetro suficiente para que la electrónica utilizada entre dentro de él.

La parte más alta del tubo, deberá ser más ancha que el resto, mantendrá al microcontrolador maestro con la placa PyTrack, la antena y la batería flotando en la superficie del agua. Los demás microcontroladores se encontrarán a diferentes profundidades para tomar diferentes medidas.

El sensor de temperatura de cada microcontrolador esclavo deberá salir de esta cubierta para realizar las medidas correctamente por lo que se creará un agujero y se sellará con una silicona. El sensor de luz deberá colocarse de cara a la superficie del agua a la misma altura que su sensor de temperatura correspondiente.

7. Protocolos de comunicación

En este apartado se estudiarán los diferentes métodos de comunicación posibles en el módulo y se elegirá el más idóneo.

El **sensor óptico** se tiene que comunicar con su correspondiente microcontrolador con el **protocolo I^2C**. La principal característica del I^2C es que utiliza dos líneas para transmitir la información: una para los datos (SDA) y otra para la señal de reloj (SCL). También es necesaria una tercera línea, la referencia (GND). Como suelen comunicarse circuitos en una misma placa que comparten una misma masa esta tercera línea no suele ser necesaria.

Este hecho limita su uso únicamente a entornos de poca interferencia, en los cuales no se ha de esperar ningún tipo de ruido, problemas de compatibilidad electromagnética o diafonías, ni problemas de contacto (clavijas, enchufes). En nuestro caso estos problemas son mínimos.

Así mismo, no es adecuado utilizar este protocolo para grandes distancias. Este es el motivo por el cual no se utiliza para comunicar el microcontrolador principal con los secundarios.

La comunicación entre el **sensor de temperatura** y su correspondiente microcontrolador utiliza el protocolo **Dallas 1-Wire**. Utiliza una única línea de cableado para la transmisión de datos digitales (Adafruit, Adafruit, s.f.).

El envío de los datos del **microcontrolador principal a la nube** podría ser por vía **LoRa o SigFox**. Estas redes se adaptan perfectamente al caso de uso IoT, donde los objetos conectados envían pequeñas cantidades de datos generados por el sensor y funcionan con la energía de la batería.

LoRa es un estándar patentado de tecnología inalámbrica de largo alcance que opera en el espectro de frecuencias radioeléctricas: 863 a 870 MHz en EU (902 a 928 MHz en Estados Unidos). Es un protocolo de capa PHYsical (OSI Layer 1) que ofrece un medio de comunicación de largo alcance y baja potencia para aplicaciones de máquina a máquina (M^2M) e IoT. A diferencia de SigFox, los módulos estándar LoRa pueden funcionar de forma bidireccional. Usando el mismo módulo de radio, un receptor puede hacer de transmisor. En el caso del KdUINO Pro no es una funcionalidad necesaria.

Por otro lado, **SigFox** es una tecnología Ultra-NarrowBand que funciona con frecuencias de sub-GHz en bandas de radio: 868MHz en Europa / ETSI y 902MHz en Estados Unidos / FCC. En esta red los dispositivos envían sus datos a través de la red SigFox hacia un *Backend Sigfox* y este maneja la transferencia de los mensajes haciendo una petición HTTP a un backend preconfigurado. En mayo de 2017 la red abarca 33 países (cobertura), la

compañía despliega sus antenas con la ayuda de compañías locales de telecomunicaciones alrededor del mundo, de modo que el proveedor IoT no tiene que preocuparse por la creación de la infraestructura. Existe una serie de restricciones en los mensajes transferidos a través de la red, donde cada dispositivo puede enviar sólo 140 mensajes por día con un límite de 7 mensajes cada hora y cada mensaje puede tener hasta 12 bytes de longitud (Gracia, 2017).

Figura 14: De izquierda a derecha, Logo de SigFox y LoRa (Gracia, 2017).

En este proyecto se ha optado por utilizar **SigFox** como red de comunicación entre microcontrolador maestro y usuario. No existe una red mejor que otra para este caso, tanto LoRa como SigFox ofrecen una buena cobertura en España. La elección entre uno u otro se debería realizar en función de la **cobertura** de cada protocolo y la **experiencia personal** que se tenga con cada uno de ellos.

El proceso de enviar datos desde el microcontrolador maestro vía SigFox, se lleva a cabo mediante 3 mensajes: **GPS, datos de luz, datos de temperatura**. Teniendo en cuenta el tipo de mensaje, la codificación será diferente, así como la descodificación a la hora de extraer los datos.

El Instituto de Ciencias del Mar usó en el proyecto KdUINO la codificación de GPS y de los datos de luz de la manera que se explicará a continuación. Al no entrar dentro del alcance de este proyecto, la codificación no se ha optimizado para comprimir los datos, pero se pretende hacer en un futuro.

Figura 15: Mensaje de datos GPS envíado vía SigFox (12 bytes)

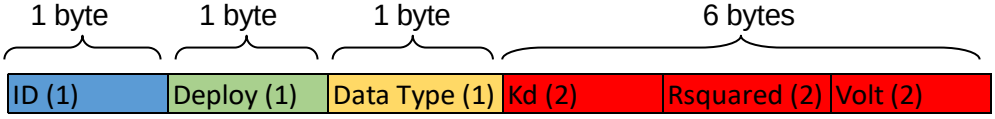


Figura 16:Mensaje de datos de luz de baja precisión envíado vía SigFox (12 bytes)

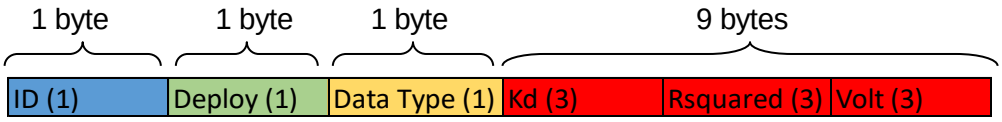


Figura 17: Mensaje de datos de luz de alta precisión envíado vía SigFox (12 bytes)

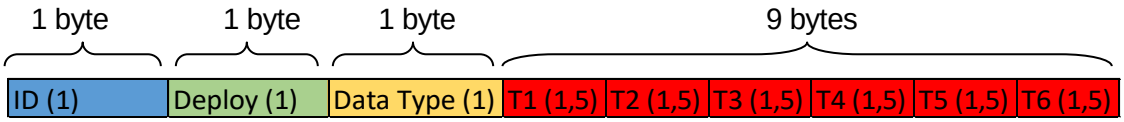


Figura 18: Mensaje de datos de temperatura envíado vía SigFox (12 bytes)

Los mensajes recogen los siguientes datos:

- ID: Identificador del módulo KdUINO Pro.
- Deploy: Mensaje que permite ordenar en el tiempo los datos recogidos.
- Data Type: El tipo 1 corresponde a GPS, 2 a datos de luz de baja precisión, 3 a datos de luz de alta precisión, 4 a datos de temperatura, sirve para realizar la descodificación correcta.
- Lat, Lon, Alt, Hdop: Latitud, longitud, altitud y calidad de la señal GPS, respectivamente.
- Kd, Rsquared: Parámetro Kd y coeficiente de determinación R^2.
- Volt: Voltaje
- T1, T2, T3, T4, T5, T6: Temperaturas, de menor a mayor profundidad, recogidas por los primeros 6 esclavos. Si existen menos de 6 esclavos, el valor inexistente será igual a 0

Como se explicará más adelante en el apartado *9. Programación*, el usuario decidirá si se mandan los datos de luz con una mayor (12 bytes) o menor precisión (9 bytes). Los dos mensajes no pueden ser enviados en un mismo script.

A la hora de elegir el protocolo de **comunicación entre el módulo principal y las unidades independientes** se ha de tener en cuenta dos factores principalmente:

- **Distancia** entre dispositivos (la distancia máxima se establece en 40m)
- **Ratio de transferencia de datos** (que no sea demasiado bajo, aunque no es primordial).

En los siguientes apartados se estudiarán dos opciones de comunicación entre los microcontroladores: inalámbricas y por cable.

7.1. Comunicación inalámbrica

El hecho de realizar una comunicación inalámbrica tiene varias ventajas que la harían la opción óptima para implementar la comunicación entre unidades de microcontroladores.

- **Elimina la necesidad de cables**. En el caso de la conexión por cable, es posible que se necesiten unos cables especiales preparados para soportar las condiciones de su entorno o bien cubrirlos con una cubierta protectora. Estos cables o carcasa elevan el precio, por lo que eliminar esta dependencia sería un punto muy favorable.
- **Minimizar mantenimiento y reparación**. Al no depender del cableado, el mantenimiento se limitaría a asegurar la distancia y posibles mejoras o cambios de código.
- **Facilidad** de realizar la comunicación. La comunicación inalámbrica se encuentra muy trabajada y existe mucho conocimiento e información al respecto.

Por otro lado, al encontrarnos en un entorno marino, existe una limitación en cuanto a la información sobre cómo se comportará esta comunicación en estas condiciones.

Uno de los principales problemas en la comunicación bajo el agua es el bajo ratio de datos disponible debido al uso de frecuencias bajas. Además, hay varios inconvenientes inherentes al medio como las reflexiones, refracciones, dispersión de la energía, etc., que degradan la comunicación entre dispositivos. En muchos casos, los dispositivos deben colocarse demasiado cerca, lo cual no interesa en este proyecto.

Mientras que normalmente se concentran los esfuerzos a incrementar el ratio de datos en bajas frecuencias, en 2012, se realizó un estudio de la comunicación inalámbrica bajo el mar en la banda de frecuencia de 2,4GHz, específicamente en el rango entre 2.412 GHz y 2.442 GHz.

Desde el punto de vista de las aplicaciones, se puede pensar que la comunicación en la banda de 2.4GHz es poco práctico debido a la alta atenuación del agua a estas frecuencias. Sin embargo, en el artículo mencionado se demuestra que existen varias aplicaciones usando ondas electromagnéticas *(Lloret, Sendra, Ardid, & Rodrigues, 2012)*.

En el artículo mencionado se estudiaron los estándares mostrados en la siguiente tabla buscando un compromiso entre bajo consumo y un ratio de datos suficientemente alto.

Tabla 4: Protocolos usados en el experimento, frecuencia y ratio de datos (Lloret, Sendra, Ardid, & Rodrigues, 2012)

Standard	Frequency	Data Rate
IEEE 802.11b	2.4 GHz	11 Mbps
IEEE 802.11g	2.4 GHz	54 Mbps
IEEE 802.15.4	2.4 GHz	250 kbps
IEEE 802.15.4	868/915 MHz	40 kbps

Este estudio consiguió unos resultados en la distancia máxima entre sensores, el número de paquetes perdidos y el tiempo de transferencia medio de datos. Aunque el ratio de transferencia de datos conseguido fue elevado, la distancia era demasiado pequeña.

En la *Tabla 5* se comparan 3 tipos de tecnología en comunicaciones inalámbricas (**ondas acústicas, ondas electromagnéticas y ondas ópticas**).

Tabla 5: Tabla comparativa de diferentes tecnologías de comunicación inalámbrica en diferentes condiciones (Lloret, Sendra, Ardid, & Rodrigues, 2012)

Technology	Working frequency	Length Wave	Modulation	Distance	Data transfer rates
ElectroMagnetic waves	3 KHz	N/app	N/av	40 m	100 bps
ElectroMagnetic waves	100 KHz	N/app	BPSK	6 m	1 Kbps
ElectroMagnetic waves	10 KHz	N/app	BPSK	16 m	1 Kbps
ElectroMagnetic waves	1 KHz	N/app	BPSK	2 m	1 Kbps
Optical Waves	N/av	420 nm	PPM	1.8 m	100 Kbps
Acoustic Waves	800 KHz	N/app	BPSK	1 m	80 Kbps
ElectroMagnetic waves	100 MHz	N/app	N/av	0.053 m	N/av
Acoustic Waves	12 KHz	N/app	MIMO-OFDM	N/av	24.36 Kbps
Acoustic Waves	24 KHz	N/app	QPSK	2500 m	30 Kbps
ElectroMagnetic waves	25 MHz	N/app	N/av	85 m	N/av
ElectroMagnetic waves	5 MHz	N/app	N/av	90 m	500 Kbps
Optical Waves	N/av	N/app	N/av	11 m	9.69 Mbps
Optical Waves	N/av	470 nm	N/av	10 m	10 Mbps
Acoustic Waves	70 KHz	N/app	ASK	70 m	0.2 Kbps
ElectroMagnetic waves	2.4 GHz (ISM Band)	N/app	BPSK	0.17 m	1 Mbps
ElectroMagnetic waves	2.4 GHz (ISM Band)	N/app	QPSK	0.17 m	2 Mbps
ElectroMagnetic waves	2.4 GHz (ISM Band)	N/app	CCK	0.16 m	5.5 Mbps
ElectroMagnetic waves	2.4 GHz (ISM Band)	N/app	CCK	0.16 m	11 Mbps

Note: N/app: Not applicable, N/av: Not available.

Como se puede observar, las **ondas electromágnéticas** ofrecen un ratio de transferencia de datos muy elevado a una frecuencia de 2,4 GHz comparando con las demás (dependiendo de la distancia, entre 1Mbps y 11Mbps), pero la distancia límite entre nodos es muy baja (entre 15cm y 17cm), por lo que quedan descartadas.

Si la frecuencia de trabajo es menor, el ratio de transferencia de datos es menor y también disminuye conforme aumenta la distancia entre nodos. Por otra parte, estas condiciones de trabajo consumen mucho por lo que también se descarta.

Las **ondas ópticas** llegan a tener un ratio de transferencia de datos elevado (entorno a 10Mbps) ofreciendo una distancia de hasta 11m. Estos valores podrían encontrarse en el rango buscado. Sin embargo, esta opción no se considerará debido a que el experimento fue realizado aumentando considerablemente el consumo de los dispositivos.

Las **ondas acústicas** ofrecen un ratio de transferencia de datos bajo mientras que las distancias son mucho más elevadas, pero también consumen demasiado. Al igual que las demás formas de comunicación inalámbrica, se descarta como opción.

Con este estudio, se concluye que la comunicación inalámbrica bajo el agua no es posible atendiendo a las necesidades de este proyecto. En todo caso, se podría invertir más esfuerzo en experimentos que encuentren una manera de comunicar unidades de forma inalámbrica, pero no se encuentra dentro del alcance de este proyecto y no asegura unos resultados óptimos.

7.2. Conexión por cable

La comunicación por cable ofrece ciertas ventajas frente a la comunicación inalámbrica:

- **Rápida detección de fallos**: los fallos en la comunicación son físicos y se pueden analizar a simple vista.
- **Velocidad de transmisión de datos mayor**
- **Programación más sencilla**. Si una persona es un programador con poca experiencia seguramente se sienta más cómodo con una conexión por cableado que inalámbrica. El proyecto, al ser DIY, buscará simplicidad en el diseño.
- **Seguridad** debido a que se imposibilita la adquisición de datos por parte de terceros.

Para comunicar el microcontrolador principal con las unidades independientes, tras descartar la comunicación I^2C por limitaciones de distancia entre nodos, se estudió utilizar el protocolo **Modbus.**

El estándar Modbus define un protocolo de mensajería de capa de aplicación, ubicado en el nivel 7 del modelo OSI que proporciona comunicaciones cliente/servidor entre dispositivos conectados en diferentes tipos de buses o redes. También define un protocolo en la línea serie para intercambiar datos entre un maestro y uno o varios esclavos *(MODBUS over Serial Line. Specification and Implementation Guide V1.02, 2006)*. Este último, ubicado en los niveles 1 y 2 del modelo OSI, es el que interesa en este proyecto.

Layer	ISO/OSI Model	
7	Application	MODBUS Application Protocol
6	Presentation	Empty
5	Session	Empty
4	Transport	Empty
3	Network	Empty
2	Data Link	MODBUS Serial Line Protocol
1	Physical	EIA/TIA-485 (or EIA/TIA-232)

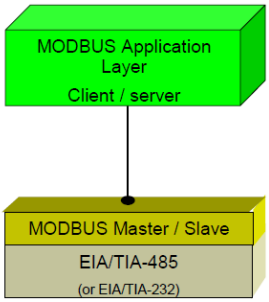


Figura 19: Protocolos Modbus y Modelo ISO/OSI (MODBUS over Serial Line. Specification and Implementation Guide V1.02, 2006)

Una comunicación Modbus siempre es iniciada por el maestro, el cual no puede soportar más de una transacción a la vez. Los esclavos no se comunicarán entre ellos y nunca transmitirán datos sin recibir la petición por parte del Maestro.

Las siguientes figuras muestran la lógica del comportamiento de cada nodo maestro y esclavo.

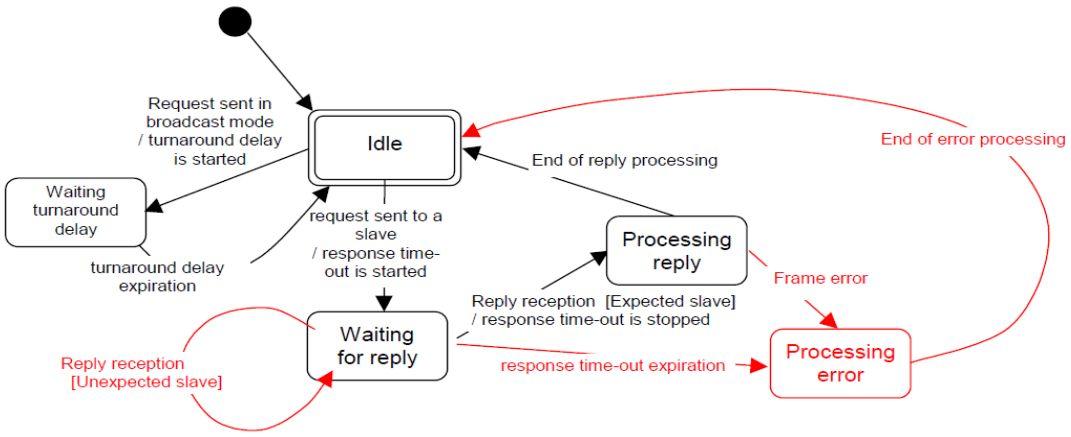


Figura 20: Diagrama de estados del comportamiento del Maestro (MODBUS over Serial Line. Specification and Implementation Guide V1.02, 2006).

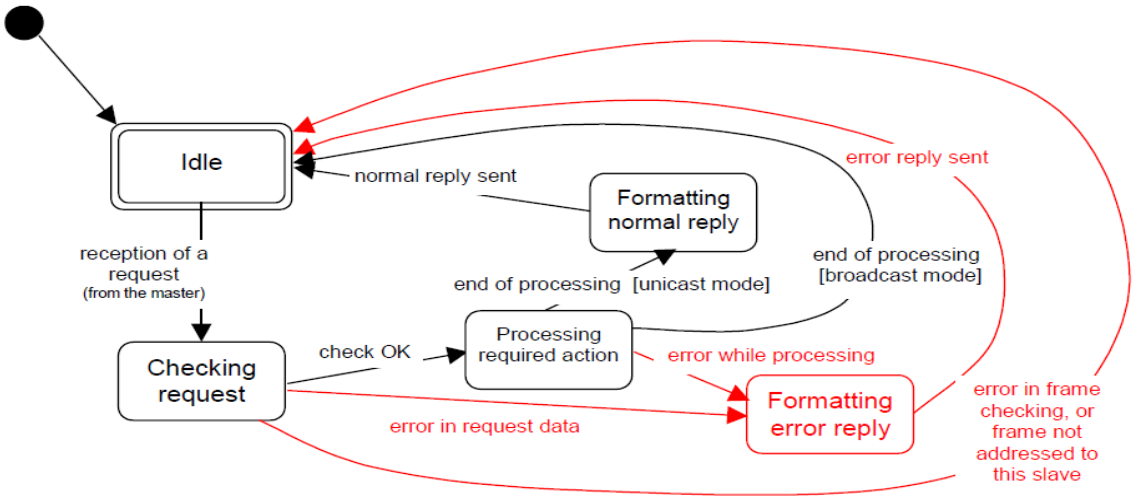


Figura 21: Diagrama de estados del comportamiento del Esclavo (MODBUS over Serial Line. Specification and Implementation Guide V1.02, 2006).

Para más información sobre cómo funciona el protocolo Modbus, consultar la guía *(MODBUS over Serial Line. Specification and Implementation Guide V1.02, 2006).*

Normalmente en las comunicaciones digitales, se trata la información con una referencia a masa, comparando una línea de masa con voltaje. Si la lectura es de 0V, será un 0 lógico (o 1 si la lógica es inversa), y si es mayor que 0V (1.8V, 3.3V o 5V), se tratará como un 1 lógico (o 0 si la lógica es inversa).

El estándar físico **RS-485**, o EIA-485, es una forma de comunicación serie, altamente usado en la industria, que no utiliza la masa como referencia, si no que utiliza la diferencia entre la tensión de dos cables $|V_B - V_A|$ (ambos a tensión mayor que la de masa).

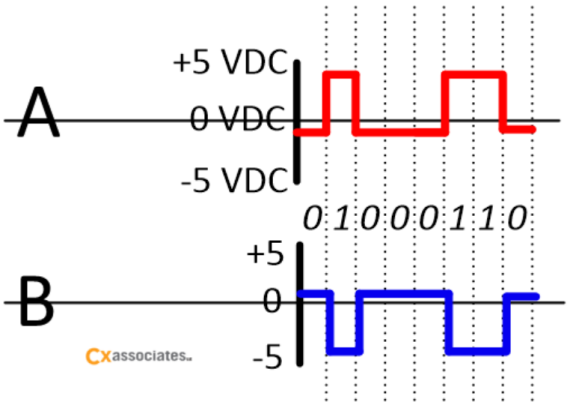


Figura 22: Lógica del estándar RS485 (Stehmeyer, 2016)

Una de las ventajas del protocolo RS-485 es que cuando una **interferencia electromagnética** afecta a las líneas de comunicación, las dos tensiones se verán afectadas por igual, y la diferencia de voltaje se mantendrá intacta. En otro tipo de comunicación, esta interferencia no afectaría a la masa, pero sí al nivel de tensión mayor a 0. De esta manera, la lectura será correcta a pesar de las interferencias que rodea a la electrónica.

El módulo será **Half duplex**, esto quiere decir que no se podrá recibir y transmitir información al mismo tiempo. Para poder hacer las dos funciones a la vez, debería ser Duplex. Esta última opción no es viable debido a las restricciones de cable que se han impuesto (solamente 4 cables).

El módulo necesitará 4 líneas: línea **A** (o X), línea **B** (o Y), **Vcc** (tensión), **GND** (masa).

Otra opción considerada fue el protocolo RS-232. Su comportamiento es similar al RS-485 pero, como se recogerá en la siguiente tabla, tiene limitaciones que impiden su utilización (Gómez, 2016):

Tabla 6: Comparación entre estándar RS232 y RS485

	RS-232	RS-485
Dispositivos conectables	1	32
Longitud de cable	15m	1200m
Velocidad máxima de transmisión	20Kbit/s	10Mbit/s
Ruido	Mayor	Menor

Como primera opción, se intentó implementar la comunicación entre microcontroladores utilizando el protocolo Modbus. Se realizaron varios intentos fallidos debido a que la biblioteca de funciones ofertada por Pycom se encuentra incompleta y poco clara, y el protocolo es lo suficientemente complicado como para implementarlo de cero.

Tras descartar Modbus, se intentó vía **UART** también usado por Modbus. UART (Universal Asynchronous Receiver Transmitter) es un dispositivo que controla los puertos y dispositivos serie. Se encuentra integrado en la placa base o en la tarjeta adaptadora del dispositivo.

UART es comúnmente usado para transmitir datos en serie. Siendo asíncrono, no existe señal de reloj, pero la estructura de los datos transmitidos dota de una secuencia de comienzo y final del mensaje. Es importante que los microcontroladores que se quieren comunicar operen con el mismo ancho de pulso definido como velocidad de transmisión (baudrate). El UART normalmente opera a niveles de 3,3V o 5V (FTDI, 2007).

La comunicación se generará a partir de tres señales. La línea **RXD** es la señal que recibe datos y se tratará como una entrada. Mientras que la línea **TXD** es la señal que transmite datos y se tratará como una salida. La elección de si el microcontrolador se encuentra transmitiendo o recibiendo datos se realiza con la señal **RE.** Cuando el nivel de tensión es alto, está transmitiendo y cuando el nivel de tensión es bajo, está recibiendo.

El problema del UART es que **no es capaz de mandar mensajes en modo broadcasting** si no se implementa una compleja función, es decir, que el maestro tendrá que mandar un mensaje a cada esclavo uno a uno. Este problema se soluciona haciendo que una vez que el esclavo haya leído el mensaje de confirmación individual enviado por parte del maestro, deje de leer datos durante cierto tiempo para que los demás esclavos puedan recibir esa confirmación individual.

En este proyecto se han realizado dos tipos diferentes de mensajes intercambiados en la comunicación del KdUINO Pro.

El **primer tipo de mensaje** se utiliza para que el esclavo mande una solicitud para entrar en comunicación con el maestro y para que este confirme esta solicitud con un nuevo mensaje de vuelta. Este mensaje será de **2 bytes** y contendrá el **identificador** (ID), único de cada esclavo.

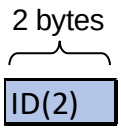


Figura 23: Mensaje de solicitud y confirmación en la comunicación UART.

El **segundo tipo de mensaje** se utiliza para que el esclavo transmita los datos almacenados al maestro. Este tipo de mensaje será de **36 bytes** y contendrá el **identificador** de el esclavo que transmite (ID), la medida de **temperatura** (Temp), **4 medidas de luz** (R, G, B y Clear) y **4 valores del logaritmo** de las medidas anteriores (Log(R), Log(G), Log(B) y Log(Clear)).

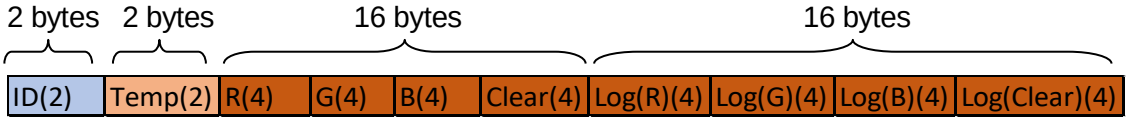


Figura 24: Mensaje de envío de datos del esclavo al maestro en la comunicación UART

Se necesitan 4 bytes para cada medida de luz y logaritmo porque el valor es de tipo coma flotante (float) mientras que el identificador y la temperatura son enteros de 2 bytes (int).

8. Conexiones

El microcontrolador LoPy4 consta de 28 pines, la mayoría de ellos son configurables vía software. Esto quiere decir que las conexiones elegidas en este proyecto podrían ser variadas siempre que el microcontrolador lo permita. Los únicos pines usados que no se pueden cambiar son los de Vcc, Tierra y 3,3V.

Si se define como unidad independiente al conjunto microcontrolador LoPy4, transceptor MAX485, y los sensores asociados (maestro no tendrá), se podría decir que las unidades se unen mediante 4 cables: línea **A, B, Vcc y Tierra**. Las líneas A y B permiten a los transceptores MAX485 traducir los datos en serie al estándar RS485.

A continuación, se mostrará una tabla con los pines asociados al microcontrolador de diferentes funciones:

Tabla 7: Conexión de pines LoPy4

Función	Pin
Vcc	Vcc
GND	GND
3.3V	3.3V
DI	P3
RO	P4
RE/DE (Cortocircuitados)	P8
SDA	P9
SCL	P21
Temperature wire	P20

Los pines **3.3V, SDA, SCL** (comunicación I^2C del sensor de luz) y **Temperature wire** (sensor de temperatura) solo se usarán en los esclavos. El sensor de temperatura DS18B20 necesita una resistencia pull-up de 4.7kΩ.

La batería irá conectada a la placa de expansión PyTrack que únicamente va conectada al LoPy4 maestro. Lo mismo sucederá con la antena SigFox (tener cuidado dónde se conecta, debe ser a la derecha del LED si éste está ubicado arriba).

Los pines **RE/DE** irán cortocircuitados. Cuando el nivel lógico sea bajo, se permitirá la escucha a través del pin asociado a **RO**, y si el nivel lógico es alto, se permitirá la escritura del pin asociado a **DI**.

En la *Figura 25* no se ha podido indicar los pines por claridad y visualización del esquema, hará falta comprobar la *Tabla 7* a la hora de conectar los pines del LoPy4. (Pycom, s.f.).

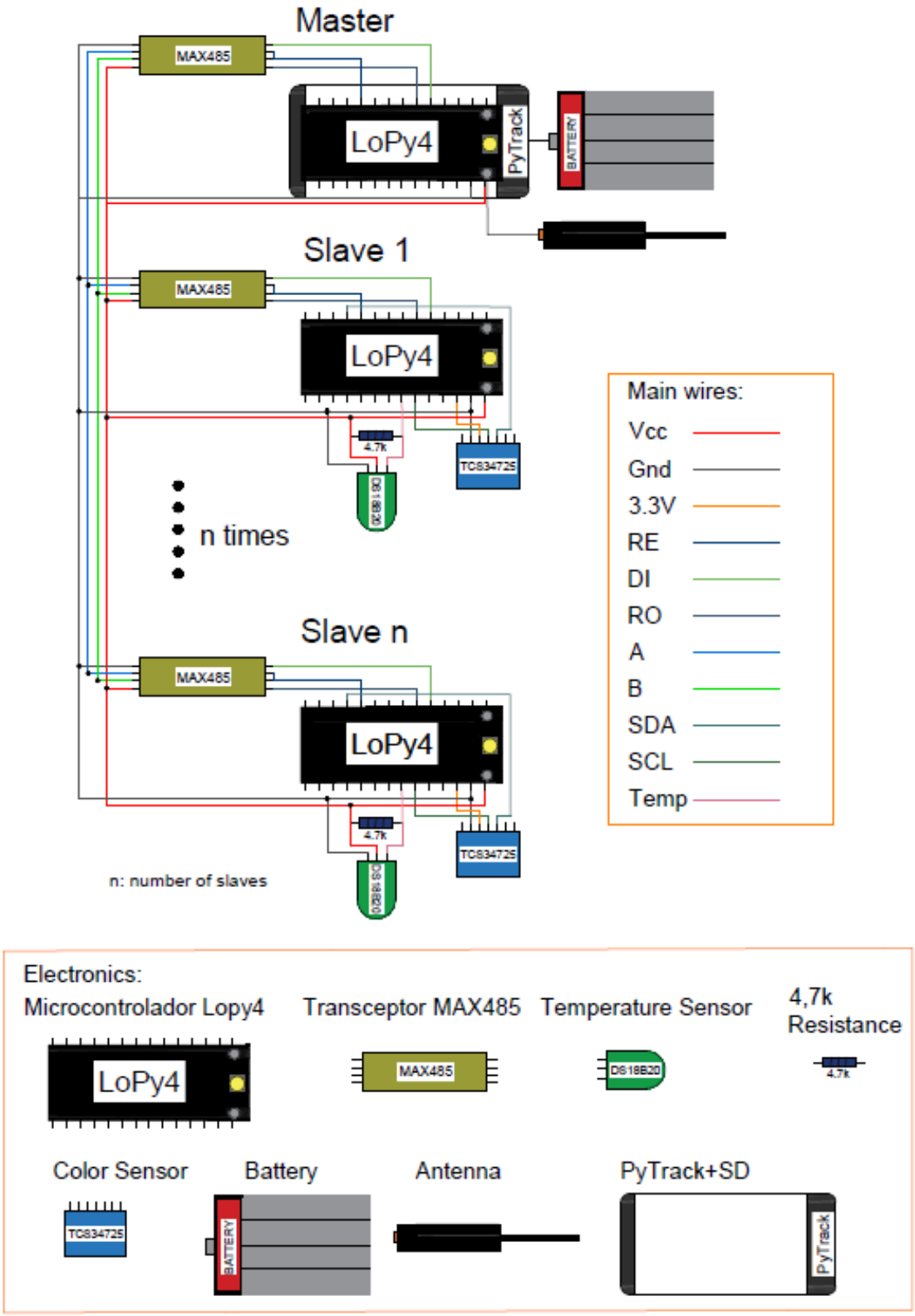


Figura 25: Diagrama de conexiones KdUINO Pro

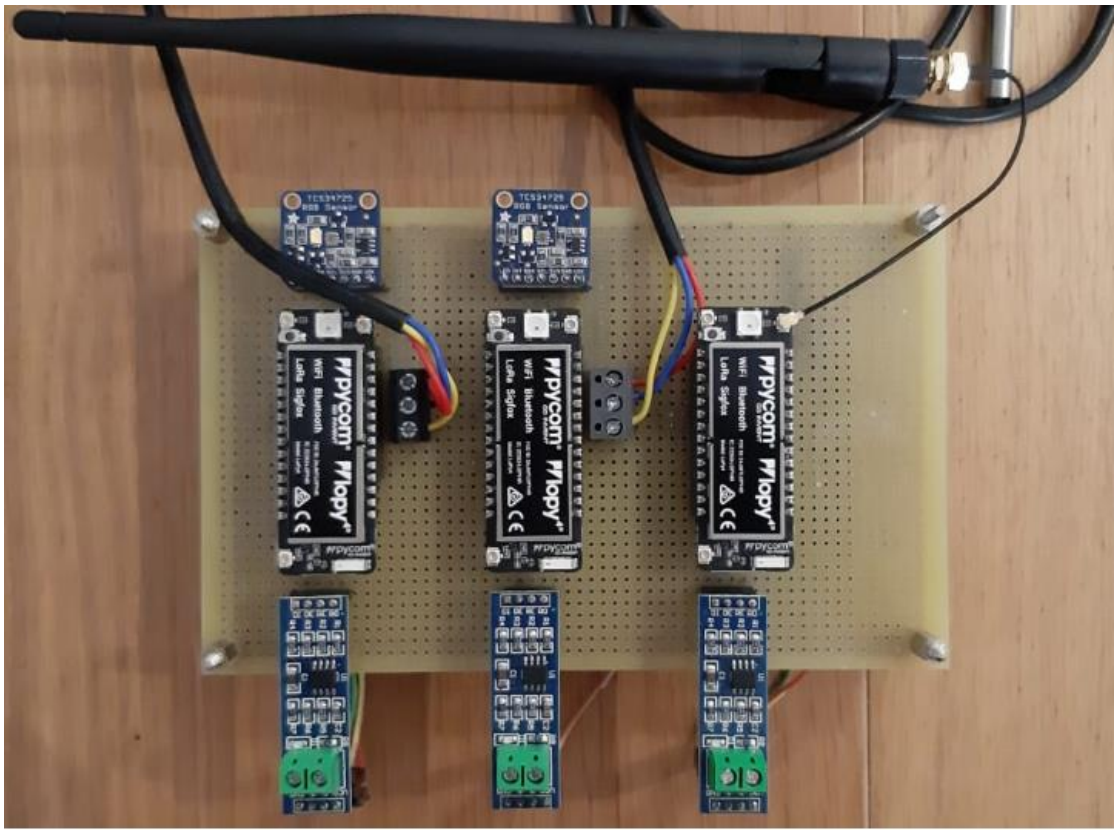


Figura 26: Placa de pruebas KdUINO Pro, vista desde arriba

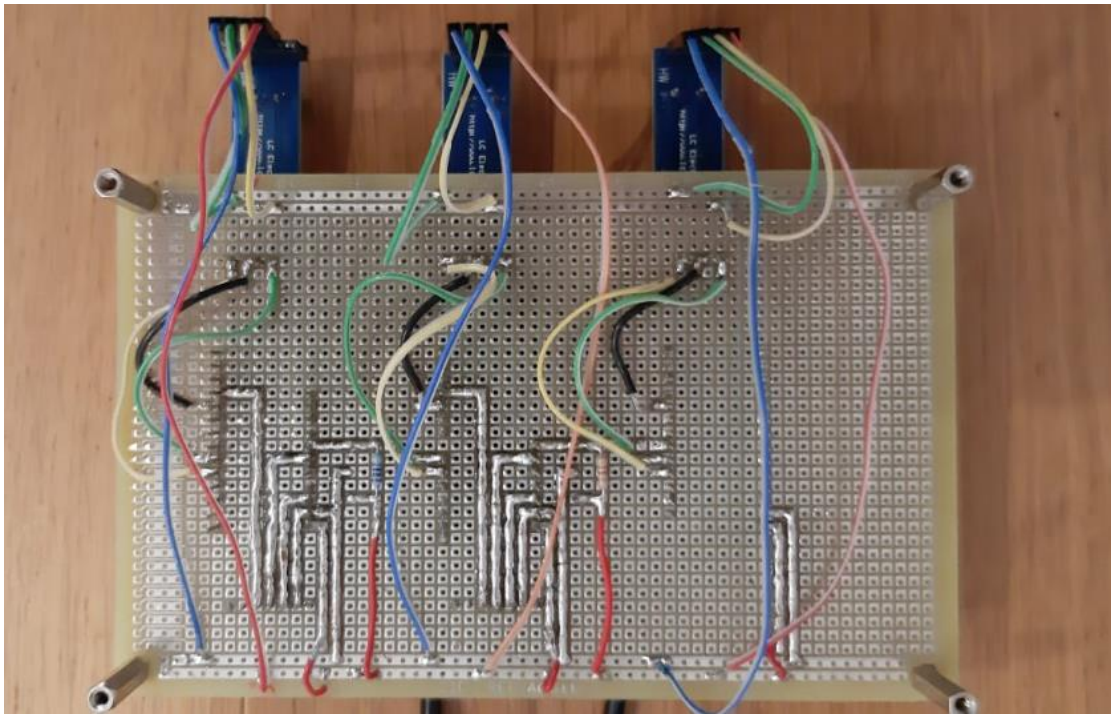


Figura 27: Placa de pruebas KdUINO Pro, vista desde abajo

9. Programación

9.1. Entornos de programación

Antes de hablar de la programación, cabe mencionar ciertos preparativos previos a la elaboración del código.

Para poder elaborar el código, fue necesario instalar ciertos programas en el ordenador:

- **VisualStudioCode**: Entorno de programación en MicroPython compatible con Pycom. También se podría haber usado Atom. Dentro de VisualStudioCode, hay que instalar PyMakr que permite interpretar correctamente los dispositivos Pycom.

- **DB Browser**: Programa que permitirá mostrar en la pantalla del ordenador los datos enviados vía SigFox por parte del módulo en varias tablas configurables.

- **Zadig**: Aplicación para instalar drivers USB. Utilizado a la hora de instalar el firmware del microcontrolador LoPy4.

Para poder escribir el código elaborado en el programa VisualStudioCode, hay que instalar previamente el firmware del microcontrolador con ayuda de Zadig. Los pasos a seguir en la instalación pueden ser confusos y complicados debido a que Pycom tiene una guía que se encuentra en estado de maduración (Pycom, User Guide Pycom, s.f.).

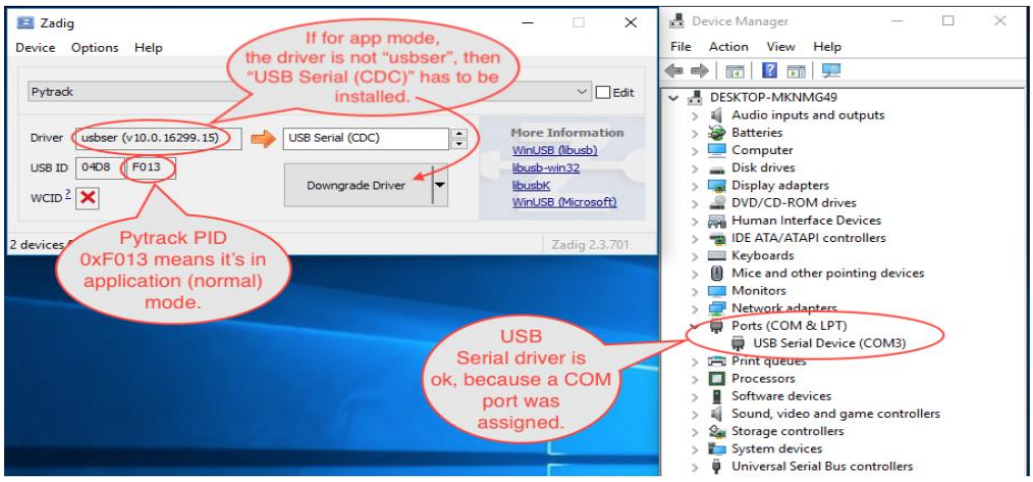


Figura 28: Instalación del firmware LoPy4 (Pycom, User Guide Pycom, s.f.)

El objetivo principal de la instalación del firmware es conseguir que el ordenador lea el microcontrolador en uno de sus puertos para poder comenzar a programar.

9.2. Librerías y funciones

Para que el código funcione correctamente, hace falta utilizar funciones que se encuentran en librerías accesibles en internet. A continuación, se mostrará una tabla que recogerá las principales librerías, su función y en qué microcontrolador se utilizará.

Tabla 8: Librerías

LIBRERÍA	FUNCIÓN	UBICACIÓN
micropyGPS	Acceder a datos del GPS de la placa PyTrack	Maestro
pycoproc	Correcto funcionamiento de la placa PyTrack	Maestro
pytrack	Funcionalidades de la placa PyTrack	Maestro
onewire	Transmisión y recepción de datos del sensor de temperatura DS18B20	Esclavos
tcs34725	Transmisión y recepción de datos del sensor de luz TCS34725	Esclavos

Al mismo tiempo, se han elaborado una serie de funciones que simplifican la ejecución del código. El funcionamiento de cada una de ellas no se explicará con diagrama de flujos (como el código principal en la *Figura 22* y *Figura 23*), debido a que es un número elevado de funciones, son complicadas y no merece la pena entrar en detalles. Si se desea entender el funcionamiento de cada una, el código se adjunta en el *Anexo 1, 2, 3 y 4*.

Tabla 9: Funciones

NOMBRE	FUNCIÓN	UBICACIÓN
blink_led	Intermitencia del led controlando el tiempo, número de intermitencias y color	Maestro y esclavos
sd_access	Acceder a la tarjeta SD	Maestro
_write_deployment	Crea un archivo "deployment" en la tarjeta SD	
_write_data	Crea un archivo "data" en la tarjeta SD	
_write_slaves	Crea un archivo "slaves" en la tarjeta SD	
get_deployment_seq	Lee y reescribe un carácter en “deployment” según la secuencia	
get_slaves	Lee y reescribe en “slaves” el número de esclavos conectados	
get_lat_lon_datetime_gps	Obtiene los datos del GPS y de la fecha	
set_datetime	Sincroniza fecha recogida con la del microcontrolador	

save_header	Escribe encabezados para la recogida de datos en la tarjeta SD	Maestro
save_data	Escribe los datos debajo de los encabezados en la tarjeta SD según el número de esclavos	
convert_data_to_payload_gps	Codifica los datos del GPS de forma que se puedan mandar en 12bytes por SigFox	
convert_data_to_low_precision_payload	Codifica los datos de transparencia de forma que se puedan mandar en 9bytes por SigFox	
convert_data_to_high_precision_payload	Codifica los datos de transparencia de forma que se puedan mandar en 12bytes por SigFox	
convert_temperature_data_to_payload	Codifica los datos de temperatura de forma que se puedan mandar en 12bytes por SigFox	
mean	Media	
best_fit_slope_and_intercept	Crea una linea aproximada con los datos recogidos del sensor de luz	
squared_error	Ayuda al cálculo del error	
coefficient_of_determination	coeficiente de determinación	
kd_rsquared	Obtiene el parámetro Kd de transparencia y el coeficiente de determinación	
read_request (master)	Maestro lee el carácter enviado por parte de los esclavos	
read_request (slave)	Esclavo lee confirmación del maestro para entrar en la comunicación	Esclavos
send_request (master)	Envía confirmación a los esclavos de que han sido aceptados en la comunicación	Maestro
send_request (slave)	Envía solicitud al maestro para entrar en la comunicación	Esclavos
read_data	Lee los datos en 32bytes enviados por parte de los esclavos	Maestro
send_data	Codifica y envía los datos en 32bytes al maestro	Esclavos
wait_requests	Espera hasta que se reciba algún valor lógico en el pin de recepción de datos en el momento de recibir solicitudes	Maestro
wait_transmision	Espera hasta que se reciba algún valor lógico en el pin de recepción de datos en el momento de recibir los datos finales	Maestro y esclavos

collect_data	Descodifica los datos recibidos en el maestro para almacenarlos en variables	Maestro
variables_automatic	Actualiza las variables en función del número de esclavos conectados	
send_data_Sigfox	Envía los datos por SigFox	
get_battery_status	Obtiene el nivel de batería actual	
deep_sleep	Sueño profundo hasta que pase cierto tiempo	Maestro y esclavos
clear_TCS34725_values	Pone a cero los valores guardados de las medidas de luz	Esclavos
clear_DS18X20_values	Pone a cero los valores guardados de las medidas de temperatura	
read_TCS34725_sensors	Lee los valores del sensor de luz	
read_DS18X20_sensors	Lee los valores del sensor de temperatura	
read_request	Lee la solicitud del maestro	
send_data	Envía datos al maestro	
sql_connection	Abre la conexión hacia la base de datos SQLite3	Ordenador
get_user_pwd_from_properties	Obtiene usuario y contraseña de SigFox	
get_and_list_jsons	Obtiene variables necesarias del backend de SigFox	
get_data	Guarda los datos en unas variables	
decoder	Descodifica los datos recibidos desde el maestro	
insert_data_to_sqlite	Crea una tabla en la base de datos SQLite e inserta los datos correspondientes	

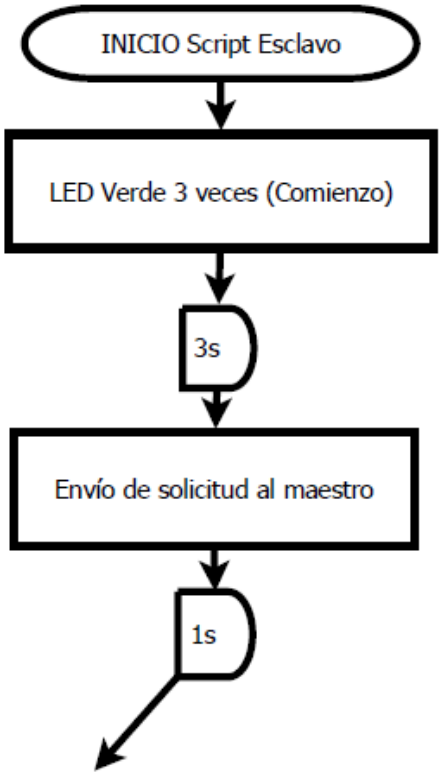
En cuanto a la parte de gestión de los datos en el backend de SigFox para poder visualizarlos en el programa DB Browser (Ordenador), solamente se han mencionado las funciones que se han de modificar para poder personalizar las tablas mostradas. El programa completo se puede encontrar en la guía de PyCom (Pycom, s.f.), por su extensión no se explica en detalle en este documento.

9.3. Código

Se comenzará explicando la programación de los **microcontroladores esclavos**. Estos microcontroladores tienen la función de recoger las medidas de los sensores, tanto de luz como de temperatura, y enviar estos datos al microcontrolador maestro cuando éste lo solicite. Una vez enviados los datos, se activa el modo "Deep-sleep" de dicho microcontrolador.

Para comprobar que todo funciona según lo esperado, se añadió una función que permitía controlar la secuencia del código mediante la realización de intermitencias en el led con diferentes colores.

Para poder entenderlo, se muestra visualmente un diagrama de flujos del código que resume muy brevemente el código realizado en los esclavos. Como ya se ha mencionado anteriormente, muchas funciones no se explican al detalle por motivos de extensión. Si se quiere entender alguna de ellas más al detalle, cada una contiene en el código una explicación de su funcionamiento.



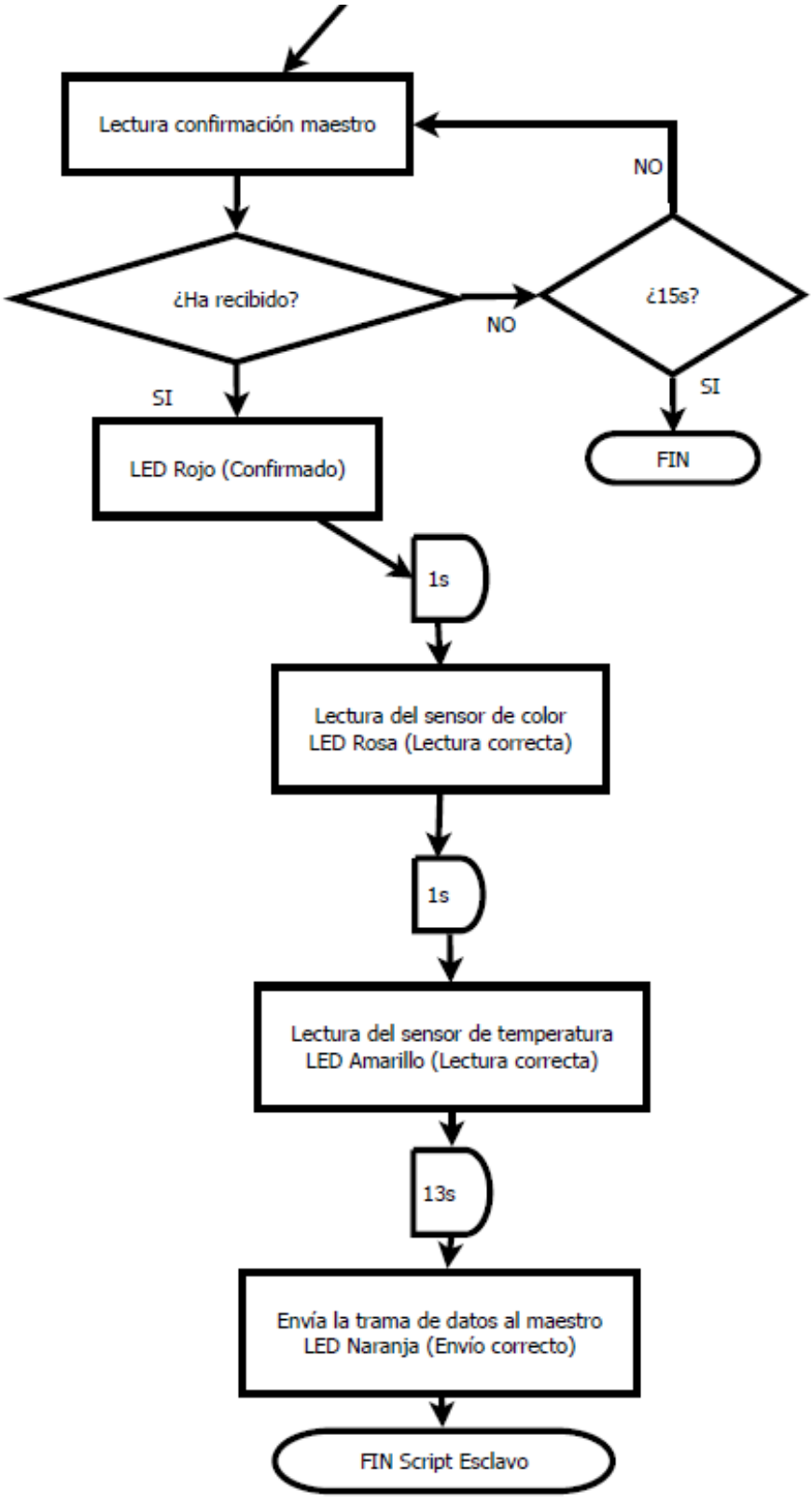
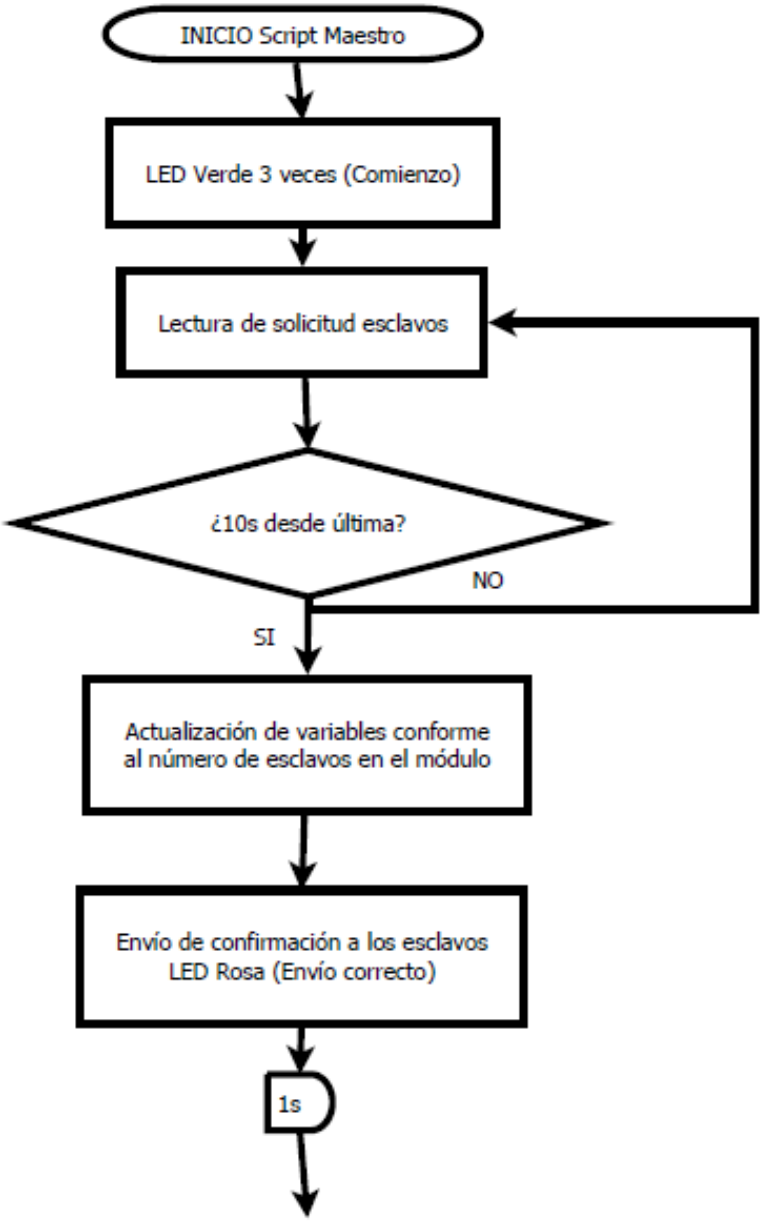
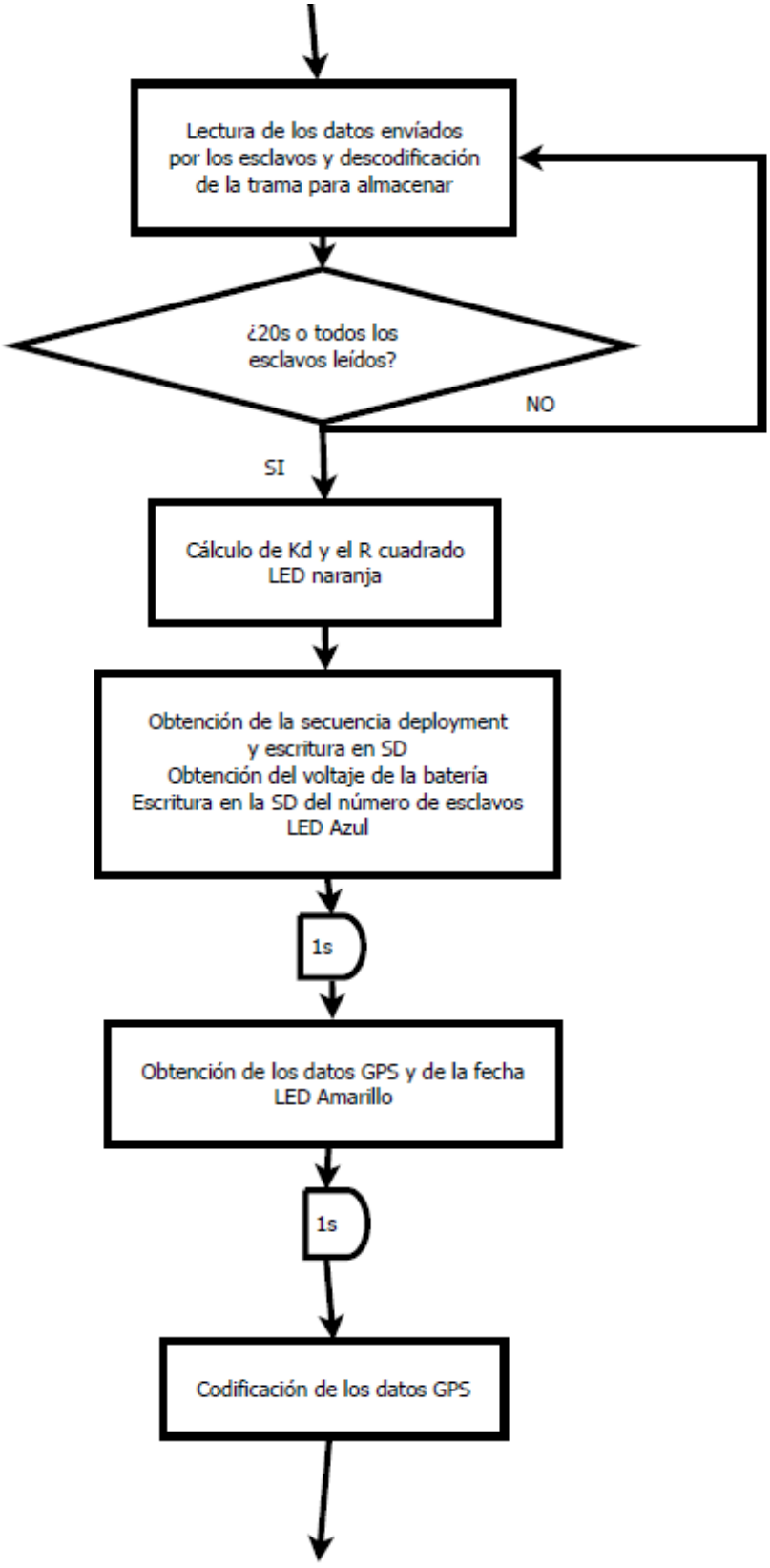


Figura 29: Diagrama de flujos del código de los esclavos

Por otro lado, el **microcontrolador maestro** recibe una solicitud por parte del esclavo y confirma a los esclavos que forman parte de la comunicación. Tras recibir todos los datos, en 32 bytes cada mensaje, de los esclavos, calcula el parámetro de transparencia Kd y el coeficiente de determinación R^2 y, junto con la temperatura y los datos del GPS, los almacena en su tarjeta SD. Al mismo tiempo, codifica estos datos de forma que quepan en mensajes de 12 bytes y se puedan enviar por SigFox.




Lectura de los datos enviados por los esclavos y descodificación de la trama para almacenar
¿20s o todos los esclavos leídos?
NO
SI
Cálculo de Kd y el R cuadrado
LED naranja
Obtención de la secuencia deployment y escritura en SD
Obtención del voltaje de la batería
Escritura en la SD del número de esclavos
LED Azul
1s
Obtención de los datos GPS y de la fecha
LED Amarillo
1s
Codificación de los datos GPS

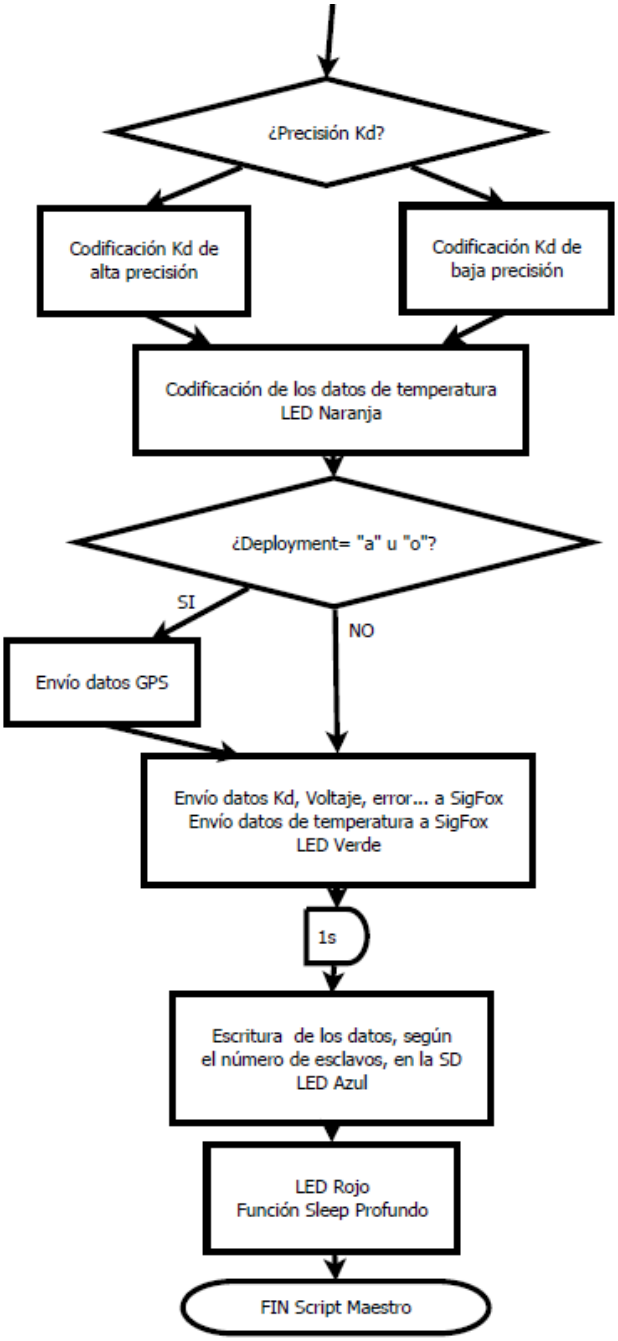


Figura 30: Diagrama de Flujos del código del maestro

Como se ha comentado previamente, no se explicará el código que almacena y descodifica los datos recogidos de SigFox y muesta los resultados en el programa DB Browser desde un ordenador. Lo único que hay que saber es que se puede modificar los encabezados de las tablas y los datos asociados, hay que descodificar el mensaje codificado por parte del maestro y que se debe poder acceder al identificador del dispositivo y su contraseña.

10. Test de validación y presupuesto

10.1. Test de validación

A lo largo del proceso de elaboración del código, se fue comprobando que el programa funcionaba según lo esperado. Esta comprobación se realizaba mediante escritura de mensajes en la pantalla de la terminal del programa Visual Studio Code, con una simulación con datos predeterminados o escritura en una tarjeta SD. Cierta parte del código también se simulaba con el programa Spyder para acelerar el proceso de cálculo sin tener que encender los microcontroladores.

Cabe mencionar previamente que todas las pruebas se han llevado a cabo en un laboratorio tomando **datos del aire**, en un entorno alrededor de 25º, y no del agua debido a que el diseño mecánico que permite aislar la electrónica del agua no se obtuvo a tiempo. Este hecho no debería condicionar el funcionamiento del módulo ya que simplemente varían los datos que se manejan, pero no su comportamiento. Además, solo se disponía de 2 microcontroladores esclavo, por lo que, si se quería simular más de 2 esclavos, uno de ellos debía mandar con un identificador distinto los datos almacenados repetidos.

Inicialmente, se comprobó que el microcontrolador esclavo recogía de forma óptima los **datos de los sensores y los almacenaba en ciertas variables** apoyándose en funciones diseñadas. Al principio no se conseguía el dato de temperatura y se debía a que la primera lectura daba un valor saturado. Al descartar este primer valor, el siguiente dato era correcto. El sensor óptico se comportaba de manera adecuada en todo momento.

Después, se pasó a comprobar que el microcontrolador maestro podía **calcular los parámetros deseados** de forma correcta con unos datos predeterminados para el test. Más tarde, se testeó si se **almacenaban en la tarjeta SD** los datos GPS, fecha, identificador de dispositivo, datos (predeterminados para el test), parámetros calculados, encabezados, etc.

Tabla 10: Información recopilada en la SD con 3 esclavos

#device_id	deployment	datetime	latitude	longitude	altitude	hdop	voltage
T	q	01/01/1970 0:00	0	0	0	0	3.627.921
T	r	01/01/1970 0:00	0	0	0	0	3.602.831

data_sent	mean	measures	depth1	depth2	depth3	temperature1	temperature2	temperature3
True	True	10	0.3	0.6	0.9	2500	2475	2475
True	True	10	0.3	0.6	0.9	2500	2475	2475

red1	green1	blue1	clear1	red2	green2	blue2	clear2	red3	green3	blue3	clear3
210.6	125.5	148.7	456.5	183.7	86.1	107.5	366.8	183.7	86.1	107.5	366.8
201.4	119.5	142.0	435.7	182.1	83.6	105.1	359.8	182.1	83.6	105.1	359.8

kd red	r squared red	kd green	r squared green	kd blue	r squared blue	kd clear	r squared clear
0.2277611	0.75	0.6279942	0.7500006	0.5407331	0.7500007	0.3646252	0.750001
0.1678943	0.75	0.595454	0.7500007	0.5015251	0.7500009	0.3190079	0.7500012

La *Tabla 10* recoge datos con el diseño terminado en el laboratorio. Al realizarse la prueba en el laboratorio los datos GPS no se recogían de manera correcta, la última prueba se realizará en un entorno abierto y se podrá observar que la función recoge unos datos correctos. Lo importante de esta prueba es observar que los datos se almacenan correctamente en la SD.

Más adelante se comprobó que al variar el número de esclavos, se reiniciaba el archivo cambiando los encabezados según esa cantidad. Se realizó la prueba con una simulación de un total de 6 esclavos. Las tablas se generaban igual que con 3 esclavos, pero añadiendo columnas para insertar datos con sus cabeceros correspondientes.

Antes de realizar la comunicación de esclavo a maestro, se comprobó que el maestro era capaz de enviar los datos por SigFox y que estos datos se podían mostrar en el programa DB Browser.

Una vez comprobado todo lo anterior, se pasó a realizar el código que permitía la comunicación entre maestro y esclavo. Este paso fue el último por la demora en conseguir los transceptores MAX485.

Se comenzó intentando establecer una comunicación con el protocolo Modbus. Debido a una limitada librería ofertada por Pycom, Modbus se descartó y se intentó realizar la comunicación vía UART diseñando las tramas de los mensajes enviados.

Inicialmente se comprobó que el maestro era capaz de recibir dos bytes codificados por parte de cada esclavo. Más tarde se añadió la función de confirmar la solicitud a cada esclavo con otros dos bytes.

Finalmente, se añadió la función de enviar los datos recibidos en los sensores por parte de cada esclavo al maestro que permanecía en la espera. Estos datos se envían en 36 bytes, como se ha comentado en el apartado *7.2 Conexión por cable*.

Al conseguir esto, se comprobó todo el módulo conjuntamente, simulando 3 esclavos, viendo que todos los resultados obtenidos en SigFox concordaban con los valores que se iban obteniendo en el terminal del ordenador y que realizaba la función Deep-Sleep y cada microcontrolador volvía a realizar el Script correspondiente.

Tabla 11:Datos GPS recogidos en SigFox

id	time	device	lqi	device_id	deployment	data_type	latitude	longitude	altitude	hdop
1568311488	12/09/2019 18:04	4D5038	EXCELLENT	T	p	0	41,400386774563	2,1523655743817	-63,00	1.5

Tabla 12: Datos de Kd, R^2 y Voltaje recogidos en SigFox en 3 medidas diferentes

id	time	device	lqi	device_id	deployment	data_type	kd	r_squared	voltage
1568311055	12/09/2019 17:57	4D5038	EXCELLENT	T	n	2	0.0299447792735563	0.749999985098838	4,6967387018644
1568311276	12/09/2019 18:01	4D5038	EXCELLENT	T	o	2	0.0526350767990991	0.749999985098838	4,7117918557997
1568311497	12/09/2019 18:04	4D5038	EXCELLENT	T	p	2	0.0688380044006109	0.749999985098838	4,70677463452665

Tabla 13: Datos de temperatura recogidos en SigFox en 3 medidas diferentes

id	time	device	lqi	device_id	deployment	data_type	temperature1	temperature2	temperature3	temperature4	temperature5	temperature6
1568311064	12/09/2019 17:57	4D5038	EXCELLENT	T	n	3	24.0	25.25	25.25	0.0	0.0	0.0
1568311284	12/09/2019 18:01	4D5038	EXCELLENT	T	o	3	23.93	25.06	25.06	0.0	0.0	0.0
1568311506	12/09/2019 18:05	4D5038	EXCELLENT	T	p	3	23.93	25.25	25.25	0.0	0.0	0.0

Los parámetros recogidos en estas tablas son:

- Id: identificador del mensaje enviado, no editable.
- Time: Fecha y hora de las tomas de datos.
- Device: Identificador del microcontrolador por parte de SigFox, no editable.
- Lqi: Calidad de la señal del mensaje de SigFox.
- Device_id: Identificador del módulo KdUINO Pro, editable.
- Deployment: carácter para identificar la secuencia de los mensajes.
- Data type: Permite decodificar de forma correcta.
- Latitud, longitud, altitud.
- Hdop: indica la calidad de la señal GPS.
- Kd y R^2.
- Voltaje.
- Temperaturas.

10.2. Presupuesto

Uno de los objetivos principales del KdUINO Pro es ofrecer una opción económica a la medida de parámetros en el medio acuático. Con un **gasto mínimo de 180,24€** (considerando 3 esclavos) se aprecia que el objetivo se ha cumplido de una forma más que satisfactoria.

Tabla 14: Presupuesto

Elemento	Precio individual (€)	Cantidad	Precio total (€)
LoPy4	24,95	2	49,9
Pytrack	30,95	1	30,95
Tarjeta SD	9,99	1	9,99
Cables	2	1	2
Sensor de luz TCS34725	4,95	1	4,95
Sensor de temperatura DS18B20	6,95	1	6,95
Transceptor MAX485	0,45	2	0,9
TOTAL 1 MAESTRO			**68,34 €**
TOTAL 1 ESCLAVO			**37,3 €**

Este presupuesto puede variar debido a la gran cantidad de opciones que existe a la hora de elegir elementos electrónicos. Lógicamente, como con todos los proveedores, si se pide por lotes, se conseguirían ofertas que rebajarían el precio de cada elemento.

10.3. Impacto ambiental

El KdUINO Pro, desde el principio se diseñó de forma que respetase el medioambiente y no pudiese dañar el entorno acuático y/o seres vivos que habitan en él. El desarrollo del prototipo se ha realizado en todo momento en un laboratorio cerrado, un **entorno controlado**.

El módulo será dirigido por la fuerza del agua, un movimiento natural, por lo que el impacto en el comportamiento de los seres vivos acuáticos no se verá alterado. Además, se buscará que su forma sea redondeada para que no quepa posibilidad de dañar a ningún animal o ser humano.

La batería permitirá al módulo recoger datos durante un largo periodo de tiempo. Esto evitará la frecuente presencia humana para recoger el módulo y recargar la batería.

Los elementos electrónicos se encontrarían **aislados del entorno** por un tubo, por lo que, el material de los elementos del interior del módulo no tiene relevancia a la hora de evaluar su impacto. El elemento con un mayor nivel de peligrosidad sería la batería. Este elemento es reutilizable y se encuentra aislado del exterior. Ningún elemento escogido contamina ni el agua salada ni dulce, pero cabría hacer un mantenimiento de la cubierta para evitar el contacto con cualquier aparato electrónico.

En caso de defecto, los elementos deben ser vertidos en un punto que sepa gestionarlos correctamente.

La recogida de datos que se realiza de forma inalámbrica desde un ordenador no tiene un impacto en el entorno. Sin embargo, la recogida del módulo una vez se haya acabado la energía de alimentación sería obligatoriamente en una embarcación. El consumo de esta embarcación y la alteración del comportamiento de los seres vivos sería el **único impacto negativo** al medioambiente.

Por otra parte, el proyecto KdUINO Pro pretende concienciar al ciudadano de la importancia de comenzar a tomar medidas del entorno acuático. De una forma económica y educativa cualquier usuario puede participar a la hora de recaudar información sobre la transparencia del agua. Uno de los varios motivos por los que la medida del coeficiente de atenuación difusa es importante, es debido a que es un indicador del grado de pureza del agua. Cuanto más contaminada se encuentre, más se alejará del valor 0 este parámetro.

En un futuro, con proyectos DIY y de open-source, la comunidad científica se encontrará mejor preparada para hacer frente a problemas medioambientales como actualmente es la contaminación del entorno acuático.

11. Propuestas de mejora

En esta primera validación de la parte electrónica del KdUINO Pro, los objetivos impuestos al inicio del proyecto han sido cumplidos. Pero como en todo prototipo, existen varias propuestas de mejora que se pretenden implementar en un futuro para poder dar por finalizado el KdUINO Pro. Las propuestas de mejora son las siguientes:

- Realizar el **diseño mecánico** del KdUINO Pro de forma estanca y que sea plegable para poder ser transportado sin problemas.
- Aprovechar la función **Deep-sleep** de una mejor manera para consumir menos.
- Realizar pruebas de **comunicación inalámbrica**.
- Invertir en más sensores (salinidad, pH, oxígeno disuelto…)
- **Optimizar el código** teniendo en cuenta las horas en las que no haya luz natural, a la noche, y así consumir menos. También establecer un límite de batería a partir del cuál el módulo solo se utilizará para mandar señal GPS.
- Realizar una **codificación más compacta**. De esta manera, se podrán ahorrar mensajes enviados por SigFox o en la comunicación UART.
- Implementar código que permita controlar las variables o simplemente visualizar los resultados desde el **smartphone**.
- Investigar medios de transporte acuáticos a los que se pueda incorporar el KdUINO Pro de manera que obtenga medidas en varios puntos diferentes sin invertir batería en un sistema de movimiento. Esta idea es complicada por las condiciones en las que se debe colocar para que las medidas sean fiables.
- Abrir un foro que permita a cualquier usuario aportar mejoras o soluciones a ciertos problemas que surjan.

Conclusiones

KdUINO Pro ha conseguido ofrecer una alternativa de bajo coste y consumo a la hora de realizar medidas de ciertos parámetros útiles en el medio acuático. El hardware propuesto es muy versátil, robusto y sencillo de implementar al necesitar un número muy reducido de componentes, con lo que se cumplen al mismo tiempo requisitos de la Ciencia Ciudadana y de la Instrumentación Oceanográfica.

Este proyecto ***open-source*** se encuentra dentro del movimiento **DIY**, un movimiento en auge que motiva al ciudadano a realizar su propio instrumento electrónico a partir de los medios de los que dispone.

Se espera que el usuario que diseñe por su cuenta un KdUINO Pro, aporte **información útil** de muestreo en zonas, quizás, no abarcadas en estudios similares por limitaciones de instrumentación y/o presupuesto.

La **codificación** ha sido realizada con una estructura fácilmente editable y entendible, de forma que cualquier usuario comprenda y modifique sus funciones según su criterio. La **electrónica** utilizada también ha sido escogida de forma que su manipulación y las conexiones sean los menos complicadas posibles.

La **comunicación** se ha diseñado con unas tramas sencillas, de forma que los mensajes eran más extensos y se consumía más. Este diseño se debe a que se ha dado más importancia a la idea de una comunicación sencilla y flexible que a otros aspectos. Se podría implementar otro tipo de comunicación sin ningún tipo de problema.

Con un gasto mínimo de 180,34€, KdUINO Pro se impone como una alternativa sólida y potente frente a instrumentos más sofisticados. Su posibilidad de **implementación de esclavos** según la necesidad del diseñador, así como **editar el código** según se desee y añadir varios sensores, dota al prototipo de una elevada **flexibilidad**.

Al ser un diseño sin ánimo de lucro, en un futuro se implementará un portal de sugerencias que permita al ciudadano colaborar en el proyecto. Esto hará que el prototipo se encuentre en **mejora continua**.

El código completo se encuentra en los Anexos o en mi repositorio online accesible desde el siguiente enlace URL: https://github.com/DiegoIraburu/KdUINOPro

Agradecimientos

Quiero agradecer a la UPC (Universidad Politécnica de Catalunya) y al ICM (Instituto de Ciencias del Mar) la oportunidad de haber podido trabajar en este proyecto como Trabajo Final de Máster. Dentro de la UPC, mencionar la ayuda de mi tutor Vicenç Parisi, sin el cuál esta colaboración no se habría dado y la persona que me ha guiado a lo largo de este trabajo.

Por último, pero no por ello menos importante, agradecer también a Raúl Bardají, Carlos Rodero y Jaume Piera, investigadores del ICM que me han tratado como a un compañero de trabajo más, me han ayudado en el laboratorio en todo momento, enseñándome muchos conceptos nuevos y ofreciéndome los instrumentos y tecnología necesarios para realizar el tránsito de su proyecto KdUINO a KdUINO Pro.

12. Bibliografía

Adafruit. (s.f.). *Adafruit*. Obtenido de https://www.adafruit.com/product/1334

Bardají, R., & Piera, J. (2013). Crowdsourcing technologies for the monitoring of the colour, transparency & fluorescence of the sea. *Citclops*.

Bardají, R., & Piera, J. (2013). Monitoring marine environments with crowdsourcing methods: Water transparency estimation using low cost technologies. *European Optical Society*.

Bardají, R., Sánchez, A.-M., Simon, C., Wernand, M., & Piera, J. (2016). Estimating the Underwater Diffuse Attenuation Coefficient with a Low-Cost Instrument: The KdUINO DIY Buoy. *Sensors*.

BiosphericalInstrumentsInc. (2011). *PRR-800 Profiling Reflectance Radiometer*. Obtenido de http://www.biospherical.com/

Consultores, C. (2014). *Indalo*. Obtenido de http://indalo.com.es/es/disco-de-secchi/129-disco-de-secchi.html

Córdoba, M. (29 de Agosto de 2011). *fisiologoi*. Obtenido de ANÁLISIS DE SEÑALES BIOELÉCTRICAS: https://www.fisiologoi.com/paginas/A_SENALES/01_ANA_SEN_INTRODUCCION.html

Espla, A. A. (2019). *ResearchGate.* Obtenido de https://www.researchgate.net/figure/Figura-1-Mapa-de-muestreo-del-Mar-Menor-dividido-en-sectores-y-sobre-el-cual-se-han_fig1_313105494

Frondriest. (2019). *Frondriest*. Obtenido de https://www.fondriest.com/li-cor-li-192-underwater-par-sensor.htm

Gómez, E. (25 de Junio de 2016). *Rincón Ingenieril*. Obtenido de http://www.rinconingenieril.es/rs485-arduino/

Gracia, L. (24 de Mayo de 2017). *BLOG DE SOFIA2 IOT PLATFORM*. Obtenido de https://about.sofia2.com/2017/05/24/sigfox-vs-lora/

Limited, F. T. (07 de Agosto de 2007). *What is UART?* Obtenido de https://www.ftdichip.com/Support/Documents/TechnicalNotes/TN_111%20What%20is%20UART.pdf

Lloret, J., Sendra, S., Ardid, M., & Rodrigues, J. (2012). Underwater Wireless Sensor Communications in the 2.4 GHz. *Sensors*.

MaximIntegrated. (s.f.). *MaximIntegrated*. Obtenido de https://www.maximintegrated.com/en/products/interface/transceivers/MAX485.html

MODBUS over Serial Line. Specification and Implementation Guide V1.02. (2006). *Modbus.org*, 7-11.

Pycom. (s.f.). *Pycom*. Obtenido de https://pycom.io/

Pycom. (s.f.). *User Guide Pycom*. Obtenido de https://docs.pycom.io/

RSHydro. (2019). *RSHydro*. Obtenido de https://www.rshydro.co.uk/water-quality-monitoring-equipment/water-quality-testing-equipment/optical-water-quality-sensors/uv-vis-spectro-radiometers/ramses-acc-vis/

Stehmeyer, R. (21 de Septiembre de 2016). *CxAssociates*. Obtenido de What is RS-485? – Part 2: https://buildingenergy.cx-associates.com/what-is-rs-485-part-2

Anexo 1: Código Esclavo

```
#Anexo 1: Código Maestro

import machine
import micropython
import pycom
import time
import gc
from math import log
import socket
import os
import ubinascii
import struct
import sys
from micropyGPS import MicropyGPS
from pytrack import Pytrack
from machine import I2C, Pin, SD, RTC, UART
from network import Sigfox

""" SET UP """

# Heartbeat LED flashes in blue colour once every 4s to signal that the system is
alive
# Turn off firmware blinking
pycom.heartbeat(False)

# Pin values
p3 = Pin('P3', Pin.OUT)
p4 = Pin('P4', Pin.IN)
p_en = Pin('P8', mode=Pin.OUT)
p_en.value(0)

# UART
uart = UART(1, baudrate=9600)
uart.init(9600, bits=8, parity=None, stop=1)

# List of Slave IDs
number_slaves = 0
slave_id1 = 97
slave_id_list = []

temperature_sensors_list =[]

# List of light sensors values and list of log sensors values
light_sensors_list = []
light_sensors_log_list = []

# Data collected
slave_addr = None
temperature_value= None
red_value = None
```

```
log_red_value = None
green_value = None
log_green_value = None
blue_value = None
log_blue_value = None
clear_value = None
log_clear_value = None

data_received = [slave_addr, temperature_value, red_value, green_value, blue_value,
clear_value
, log_red_value, log_green_value, log_blue_value, log_clear_value]

# ID of Master device. It can be any ASCII character
device_id = "T"

# Deployment sequence. Initialize to None
deployment_seq = None

# Time values in seconds of different variables
time_searching_GPS = 120
#30
time_searching_Sigfox = 120
#90
time_to_deep_sleep = 900
#720

# Boolean to save if data have been sent to Sigfox
gps_sent = False
data_sent = False
temperature_data_sent= False

# Boolean to check if we want to send data with high precision or not
precision_data = True

# List of depths
depth1 = 0.30
depths_list = []

# Dictionary to save GPS values
data_gps = {}
data_gps['latitude'] = None
data_gps['longitude'] = None
data_gps['altitude'] = None
data_gps['hdop'] = None

# Variables to save path values
sd_mount_dir = "/sd"
deployment_filename = "/sd/deployment.txt"
slaves_filename ="/sd/slaves.txt"
data_filename = "/sd/data.txt"

# Dictionaries to save Kd and r_squared values
kd_dict, r_squared_dict = {}, {}
```

```
102
103 # Number of light measures
104 number_measures = 10
105
106 # Boolean to calculate mean of light measures for each light sensor
107 sensor_mean = True
108
109 # color leds
110 led_red = 0x7f0000
111 led_green = 0x007f00
112 led_yellow = 0x7f7f00
113 led_blue = 0x00FFFF
114 led_orange = 0xFF9900
115 led_pink = 0xFF00FF
116
117 # Enable garbage collector
118 gc.enable()
119
120 # SD
121 sd = None
122
123 # RTC
124 rtc = RTC()
125
126 """ FUNCTIONS """
127
128
129 def blink_led(times, ms, color):
130 """ Function to uses the RGB LED to make a blink light
131
132 Parameters
133 ----------
134 times: int
135 number of times to blink the light.
136 ms: int
137 time in ms that will be blink the light
138 color: string
139 type of color
140 """
141 for cycles in range(times):
142 pycom.rgbled(color)
143 time.sleep_ms(int(ms/2))
144 pycom.heartbeat(False)
145 time.sleep_ms(int(ms/2))
146
147
148 def sd_access():
149 """ Function to access to SD.
150
151 Returns
152 -------
153 sd: obj
154 initialized sd object
```

```
155 """
156 try:
157 sd = SD()
158
159 except OSError as e:
160 print("Error: {}. SD card not found".format(e))
161 while True:
162 blink_led(1, 500, led_red)
163
164 os.mount(sd, sd_mount_dir)
165 return sd
166
167
168 def _write_deployment():
169 """
170 Function to write the deployment file into the SD
171 """
172 f_write = open(deployment_filename, 'w+')
173 f_write.write("{}".format("a"))
174 f_write.close()
175
176
177 def _write_slaves():
178 """
179 Function to write the deployment file into the SD
180 """
181 f_write = open(slaves_filename, 'w+')
182 f_write.write("{}".format(len(slave_id_list)))
183 f_write.close()
184
185
186 def _write_data():
187 """
188 Function to write the data file into the SD
189 """
190 f_write = open(data_filename, 'w')
191 f_write.close()
192
193
194 def get_deployment_seq():
195 """
196 Function to get the sequence deployment from deployment file.
197 It updates the character sequence from deployment file adding 1 to the
198 sequence. I.e.: a + 1 => b
199
200 Returns
201 -------
202 deployment: string
203 character sequence from deployment.txt
204 """
205 deployment = None
206
207 sd = sd_access()
```

```
208 f_read = False
209
210 try:
211 f_read = open(deployment_filename, 'r')
212 deployment = f_read.readall()
213 f_read.close()
214 except OSError as e:
215 print("Error: {}. Deployment file not found".format(e))
216
217 if f_read is False:
218 deployment = "a"
219 _write_deployment()
220
221 if deployment is None or deployment is '' or (ord(deployment) < 97) or \
222 (ord(deployment) > 122):
223 deployment = "a"
224 _write_deployment()
225
226 f_write = open(deployment_filename, 'w+')
227 next_deployment = chr(ord(deployment) + 1)
228 f_write.write("{}".format(next_deployment))
229 f_write.close()
230
231 sd.deinit()
232 os.unmount(sd_mount_dir)
233
234 return deployment
235
236
237 def get_slaves():
238 """
239 Function to get the slaves in the last program.
240 It updates the header of the data file if the number of slaves is different
241
242 Returns
243 -------
244 slaves: string
245 character sequence from deployment.txt
246 """
247 slaves = None
248
249 sd = sd_access()
250 f_read = False
251
252 try:
253 f_read = open(slaves_filename, 'r')
254 slaves = f_read.readall()
255 f_read.close()
256 except OSError as e:
257 print("Error: {}. Slaves file not found".format(e))
258
259 if f_read is False:
260 slaves = chr(len(slave_id_list))
```

```
261 _write_slaves()
262
263 if slaves is None or slaves is '' or (ord(slaves) < 1) or \
264 (ord(slaves) > 20):
265 slaves = chr(len(slave_id_list))
266 _write_slaves()
267
268 f_write = open(slaves_filename, 'w+')
269 next_slaves = chr(len(slave_id_list))
270 f_write.write("{}".format(next_slaves))
271 f_write.close()
272
273 sd.deinit()
274 os.unmount(sd_mount_dir)
275
276 return slaves
277
278
279 def get_lat_lon_datetime_gps(time_searching_GPS):
280 """
281 Function to get latitude, longitude, altitude and hdop from GPS.
282
283 Parameters
284 ----------
285 time_searching_GPS : int
286 seconds for searching the GPS
287
288 Returns
289 -------
290 last_data : dict
291 dictionary that contains datetime, latitude, longitude,
292 altitude, and hdop
293 """
294 GPS_TIMEOUT_SECS = time_searching_GPS
295 # init I2C to P21/P22
296 i2c = machine.I2C(0, mode=I2C.MASTER, pins=('P22', 'P21'))
297 # write to address of GPS
298 GPS_I2CADDR = const(0x10)
299 raw = bytearray(1)
300 i2c.writeto(GPS_I2CADDR, raw)
301 # create MicropyGPS instance. location formatting with commas
302 gps = MicropyGPS(location_formatting='dd')
303 # start a timer
304 chrono = machine.Timer.Chrono()
305 chrono.start()
306 # store results here
307 last_data = {}
308 # return from validate coordinates
309 res = False
310
311 def check_for_valid_coordinates(gps):
312 '''
313 Given a MicropyGPS object, this function checks if valid coordinate
```

```
314 data has been parsed successfully. If so, copies it over to global
315 last_data.
316
317 Parameters
318 ----------
319 gps : obj
320 MicropyGPS object
321
322 Returns
323 -------
324 bool
325 Returns True if values from GPS are not 0
326
327 '''
328 if gps.satellite_data_updated() and gps.valid:
329 # blink_led(1, 500, led_yellow)
330
331 timestamp_list = gps.timestamp
332 time = (timestamp_list[0], timestamp_list[1], int(timestamp_list[2]), 0,
0)
333
334 date_list = gps.date
335 date = (2000 + date_list[2], date_list[1], (date_list[0]))
336
337 datetime = tuple(date) + tuple(time)
338
339 lat_list = gps.latitude_string().split("°")
340 lat = float(lat_list[0])
341
342 lon_list = gps.longitude_string().split("°")
343 lon = float(lon_list[0])
344
345 alt = gps.altitude
346
347 hdop = gps.hdop
348
349 last_data['datetime'] = datetime
350 last_data['latitude'] = lat
351 last_data['longitude'] = lon
352 last_data['altitude'] = alt
353 last_data['hdop'] = hdop
354
355 if last_data['datetime'] is not 0 and last_data['latitude'] is not 0 and
last_data['longitude'] is not 0:
356 return True
357 else:
358 return False
359
360 else:
361 return False
362
363 check_for_valid_coordinates(gps)
364
```

```
365 while True:
366 # read some data from module via I2C
367 raw = i2c.readfrom(GPS_I2CADDR, 16)
368 # feed into gps object
369 for b in raw:
370 sentence = gps.update(chr(b))
371 if sentence is not None:
372 # gps successfully parsed a message from module
373 # see if we have valid coordinates
374 res = check_for_valid_coordinates(gps)
375 elapsed = chrono.read()
376
377 if elapsed > GPS_TIMEOUT_SECS:
378 break
379
380 if 'latitude' in last_data and 'longitude' in last_data and 'datetime' in
last_data:
381 i2c.deinit()
382 return last_data
383 else:
384 last_data['datetime'] = 0
385 last_data['latitude'] = 0
386 last_data['longitude'] = 0
387 last_data['altitude'] = 0
388 last_data['hdop'] = 0
389 i2c.deinit()
390 return last_data
391
392
393 def set_datetime():
394 """
395 Function to initialize real time clock from GPS in Lopy/Sipy device
396 """
397 if data_gps['datetime'] is not 0:
398 rtc.init(data_gps['datetime'])
399 else:
400 rtc.init()
401
402
403 def save_header():
404 """
405 Function to write header to data file in SD. Header contains the following
406 values:
407 device_id, deployment, datetime, latitude, longitude, altitude, hdop,
408 voltage, data_sent, mean, measures and a list of depths and temperatures.
409 """
410
411 sd = sd_access()
412 f_test = False
413
414 depths = ""
415 rgbc = ""
416 temperatures = ""
```

```
417
418 for i, value in enumerate(depths_list):
419 depths += "{},".format("depth" + str(i+1))
420 rgbc += "{},{},{},{},".format("red" + str(i+1), "green" + str(i+1), "blue" +
str(i+1),
421 "clear" + str(i+1))
422 for i, value in enumerate(temperature_sensors_list):
423 temperatures += "{},".format("temperature" + str(i+1))
424
425 try:
426 f_test = open(data_filename, 'r')
427 except OSError as e:
428 print("Error: {}. Data file not found".format(e))
429
430 if (f_test is False) or (slaves!=len(slave_id_list)):
431 _write_data()
432 f = open(data_filename, 'a')
433 header = "{},{},{},{},{},{},{},{},{},{},{},{},{},{}{}".format("#device_id",
"deployment", "datetime", "latitude",
434 "longitude",
"altitude", "hdop",
"voltage", "data_sent",
435 "mean", "measures",
depths, temperatures)
436 rgbc_header = rgbc
437 red_header = "{},{},".format("kd red", "r squared red")
438 green_header = "{},{},".format("kd green", "r squared green")
439 blue_header = "{},{},".format("kd blue", "r squared blue")
440 clear_header = "{},{},\n".format("kd clear", "r squared clear")
441 f.write(header)
442 f.write(rgbc_header)
443 f.write(red_header)
444 f.write(green_header)
445 f.write(blue_header)
446 f.write(clear_header)
447 f.close()
448
449 sd.deinit()
450 os.unmount(sd_mount_dir)
451
452
453 def save_data(device_id=None, deployment=None, datetime=None, lat=None, lon=None,
alt=None, hdop=None, volt=None,
454 data_sent=None, temperature_data_sent=None, temperature_list=None,
kd=None, r_squared=None, raw_data=None):
455 """
456 Function to save data file in SD. Data contains the following
457 values:
458 device_id, deployment, datetime, lat, lon, alt, hdop, volt, sensor_mean,
number_measures, list of depths, list of temperatures,
459 rgbc raw data, and kd - r_squared for red, green, blue and clear.
460
461 Parameters
```

```
462 ----------
463 device_id: string
464 the id of the prototype
465 deployment: string
466 the deployment sequence of the measurement
467 datetime: tuple
468 tuple that contains the real time clock
469 lat: float
470 latitude from gps
471 lon: float
472 longitude from gps
473 alt: int
474 altitude from gps
475 hdop: int
476 hdop (quality data) from gps
477 volt: int
478 voltage from battery
479 data_sent: bool
480 True if data has been sent to Sigfox
481 temperature_data_sent: bool
482 True if data has been sent to Sigfox
483 temperature_list: list
484 list of temperature values
485 kd: dict
486 dictionary with kd values
487 r_squared: dict
488 dictionary with r_squared values
489 raw_data: list
490 list of red, green, blue and clear values
491
492 """
493 sd = sd_access()
494 f = open(data_filename, 'a')
495
496 depths = ""
497 rgbc = ""
498 temperatures= ""
499
500 for i, depth in enumerate(depths_list):
501 depths += "{},".format(depth)
502
503 for i, data in enumerate(raw_data):
504 rgbc += "{},{},{},{},".format(data[0], data[1], data[2], data[3])
505
506 for i, temp in enumerate(temperature_list):
507 temperatures += "{},".format(temp)
508
509 datetime = "{}-{}-{} {}:{}:{}".format(datetime[0], datetime[1], datetime[2],
datetime[3], datetime[4], datetime[5])
510
511 data = "{},{},{},{},{},{},{},{},{},{},{},{},".format(device_id, deployment,
datetime, lat, lon, alt, hdop, volt,
512 data_sent, sensor_mean,
```

```
number_measures)
513 rgbc_raw_data = rgbc
514 red_data = "{},{},".format(kd["red"], r_squared["red"])
515 green_data = "{},{},".format(kd["green"], r_squared["green"])
516 blue_data = "{},{},".format(kd["blue"], r_squared["blue"])
517 clear_data = "{},{},\n".format(kd["clear"], r_squared["clear"])
518
519 f.write(data)
520 f.write(depths)
521 f.write(temperatures)
522 f.write(rgbc_raw_data)
523 f.write(red_data)
524 f.write(green_data)
525 f.write(blue_data)
526 f.write(clear_data)
527
528 f.close()
529 sd.deinit()
530 os.unmount(sd_mount_dir)
531
532
533 def convert_data_to_payload_gps(device_id=None, deployment=None, lat=None, lon=None,
alt=None, hdop=None):
534 """
535 Function to convert gps data to a bytes payload
536
537 Parameters
538 ----------
539 device_id: string
540 the id of the device
541 deployment: string
542 the deployment sequence of the measurement
543 lat: float
544 latitude from gps
545 lon: float
546 longitude from gps
547 alt: int
548 altitude from gps
549 hdop: int
550 hdop (quality data) from gps
551
552 Returns
553 -------
554 payload: list
555 list of data values codified in bytes
556
557 """
558 payload = []
559 latb = int(((lat + 90) / 180) * 0xFFFFFF)
560 lonb = int(((lon + 180) / 360) * 0xFFFFFF)
561 altb = int(round(float(alt), 0))
562 hdopb = int(float(hdop) * 10)
563 data_type = 0
```

```
564
565 # payload gps: id(1) deploy(1) data_type(1) lat(3) lon(3) alt(2) hdop(1) --> 12
bytes
566 payload.append(ord(device_id))
567 payload.append(ord(deployment))
568 payload.append(data_type)
569 payload.append(((latb >> 16) & 0xFF))
570 payload.append(((latb >> 8) & 0xFF))
571 payload.append((latb & 0xFF))
572 payload.append(((lonb >> 16) & 0xFF))
573 payload.append(((lonb >> 8) & 0xFF))
574 payload.append((lonb & 0xFF))
575 payload.append(((altb >> 8) & 0xFF))
576 payload.append((altb & 0xFF))
577 payload.append(hdopb & 0xFF)
578
579 return payload
580
581
582 def convert_data_to_low_precision_payload(device_id=None, deployment=None, kd=None,
r_squared=None, volt=None):
583 """
584 Function to convert data to a bytes payload with low precision
585
586 Parameters
587 ----------
588 device_id: string
589 the id of the device
590 deployment: string
591 the deployment sequence of the measurement
592 kd: float
593 value of kd clear
594 r_squared: float
595 value of r_squared clear
596 volt: float
597 voltage from battery
598
599 Returns
600 -------
601 payload: list
602 list of data values codified in bytes
603
604 """
605 payload = []
606
607 device_id = ord(device_id)
608 deployment = ord(deployment)
609 data_type = 1
610
611 kdb = int(((kd + 10) / 20) * 10000)
612 r_squaredb = int(((r_squared + 1) / 2) * 10000)
613 voltb = int((volt / 5) * 10000)
614
```

```
615 # payload 9 bytes. Low precision in Kd, r_squared and volt
616 # payload data: id(1) deploy(1) data_type(1) kd(2) rsquared(2) volt(2) --> 9 bytes
617 payload.append(device_id)
618 payload.append(deployment)
619 payload.append(data_type)
620 payload.append(((kdb >> 8) & 0xFF))
621 payload.append((kdb & 0xFF))
622 payload.append(((r_squaredb >> 8) & 0xFF))
623 payload.append((r_squaredb & 0xFF))
624 payload.append(((voltb >> 8) & 0xFF))
625 payload.append((voltb & 0xFF))
626
627 return payload
628
629
630 def convert_data_to_high_precision_payload(device_id=None, deployment=None, kd=None,
r_squared=None, volt=None):
631 """
632 Function to convert data to a bytes payload with high precision
633
634 Parameters
635 ----------
636 device_id: string
637 the id of the device
638 deployment: string
639 the deployment sequence of the measurement
640 kd: float
641 value of kd clear
642 r_squared: float
643 value of r_squared clear
644 volt: float
645 voltage from battery
646
647 Returns
648 -------
649 payload: list
650 list of data values codified in bytes
651
652 """
653 payload = []
654
655 device_id = ord(device_id)
656 deployment = ord(deployment)
657 data_type = 2
658
659 kdb = int(((kd + 10) / 20) * 0xFFFFFF)
660 r_squaredb = int(((r_squared + 1) / 2) * 0xFFFFFF)
661 voltb = int((volt / 5) * 0xFFFFFF)
662
663 # payload 12 bytes. High precision in Kd, rsquared and volt
664 # payload data: id(1) deploy(1) data_type(1) kd(3) rsquared(3) volt(3) --> 12
bytes
665 payload.append(device_id)
```

```
666 payload.append(deployment)
667 payload.append(data_type)
668 payload.append(((kdb >> 16) & 0xFF))
669 payload.append(((kdb >> 8) & 0xFF))
670 payload.append((kdb & 0xFF))
671 payload.append(((r_squaredb >> 16) & 0xFF))
672 payload.append(((r_squaredb >> 8) & 0xFF))
673 payload.append((r_squaredb & 0xFF))
674 payload.append(((voltb >> 16) & 0xFF))
675 payload.append(((voltb >> 8) & 0xFF))
676 payload.append((voltb & 0xFF))
677
678 return payload
679
680
681 def convert_temperature_data_to_payload(device_id=None, deployment=None,
temperature=None):
682 """
683 Function to convert temperature data to a bytes payload
684
685 Parameters
686 ----------
687 device_id: string
688 the id of the device
689 deployment: string
690 the deployment sequence of the measurement
691 temperature:list
692 list of temperature values
693
694 Returns
695 -------
696 payload: list
697 list of data values codified in bytes
698
699 """
700 payload = []
701
702 device_id = ord(device_id)
703 deployment = ord(deployment)
704 data_type = 3
705 byte_mixed = 0
706
707 # payload max 12 bytes. Low precision in Kd, r_squared and volt
708 # payload data: id(1) deploy(1) data_type(1) temperature(9)
(temperature1(1,5)...temperature6(1,5)) --> max 12 bytes
709 payload.append(device_id)
710 payload.append(deployment)
711 payload.append(data_type)
712 for i in range(len(temperature)):
713 if i>5:
714 return payload
715 if i%2:
716 byte_mixed += ((temperature[i]>>8) & 0xF)
```

```
717 payload.append(byte_mixed & 0xFF)
718 payload.append(temperature[i] & 0xFF)
719 else:
720 payload.append((temperature[i]>>4) & 0xFF)
721 byte_mixed=((temperature[i]) & 0xF)<<4
722 if i==(len(temperature)-1):
723 payload.append(byte_mixed & 0xFF)
724
725 return payload
726
727
728 def mean(numbers):
729 """
730 Function that calculates mean from list of numbers we enter as a parameter.
731
732 Parameters
733 ----------
734 numbers: list
735 list of numbers
736
737 Returns
738 -------
739 float
740 mean calculated from list of numbers
741
742 """
743 return float(sum(numbers)) / max(len(numbers), 1)
744
745
746 def best_fit_slope_and_intercept(xs, ys):
747 """
748 Function to calculate slope and intercept
749
750 Parameters
751 ----------
752 xs: list
753 list of x values
754 ys: list
755 list of y values
756
757 Returns
758 -------
759 m: float
760 slope of the line
761 b: float
762 intercept of the line
763 """
764 mult_xs_ys_list = []
765 mult_xs_xs_list = []
766 for i, x in enumerate(xs):
767 y = ys[i]
768 mult_xs_ys_list.append(x*y)
769 mult_xs_xs_list.append(x*x)
```

```
770
771 m = (((mean(xs)*mean(ys)) - mean(mult_xs_ys_list)) / ((mean(xs)*mean(xs)) -
mean(mult_xs_xs_list)))
772 b = mean(ys) - m*mean(xs)
773 return m, b
774
775
776 def squared_error(ys_orig, ys_line):
777 """
778 Function to calculate squarred error
779
780 Parameters
781 ----------
782 ys_orig: list
783 list of points from y origin
784 ys_line: list
785 list of values of y line
786
787 Returns
788 -------
789 float
790 squarred error calculated
791
792 """
793 subs_ys_line_ys_orig = []
794 for i, y_orig in enumerate(ys_orig):
795
796 y_line = ys_line[i]
797 subs_ys_line_ys_orig.append((y_line - y_orig) * (y_line - y_orig))
798
799
800 return sum(subs_ys_line_ys_orig)
801
802
803 def coefficient_of_determination(ys_orig, ys_line):
804 """
805 Function to calculate coefficient of determination
806
807 Parameters
808 ----------
809 ys_orig: list
810 list of points from y origin
811 ys_line: list
812 list of values of y line
813
814 Returns
815 -------
816 float
817 coefficient of determination. Otherwise returns 0
818
819 """
820 # depth list = ys_orig, regression lineal = ys_line
821 y_mean_line = []
```

```
822
823 for i in enumerate(ys_orig):
824 y_mean_line.append(mean(ys_orig))
825
826 squared_error_regr = squared_error(ys_orig, ys_line)
827 squared_error_y_mean = squared_error(ys_orig, y_mean_line)
828 try:
829 return 1 - (squared_error_regr/squared_error_y_mean)
830 except ZeroDivisionError as e:
831 return 0
832
833
834 def kd_rsquared():
835 """
836 Function to calculate kd and r-squared
837
838 Returns
839 -------
840 kd_dict, r_squared_dict: dict
841 dictionaries with kd and r-squared values
842
843 """
844
845 red_log_list = []
846 green_log_list = []
847 blue_log_list = []
848 clear_log_list = []
849
850 for values in light_sensors_log_list:
851 red_log_list.append(values[0])
852 green_log_list.append(values[1])
853 blue_log_list.append(values[2])
854 clear_log_list.append(values[3])
855
856
857 # Calculate kd and r squared for clear
858 m_clear, b_clear = best_fit_slope_and_intercept(depths_list, clear_log_list)
859 kd_clear = m_clear*(-1)
860 regression_line_clear = [(m_clear*x)+b_clear for x in depths_list]
861 r_squared_clear = coefficient_of_determination(clear_log_list,
regression_line_clear)
862 kd_dict["clear"] = kd_clear
863 r_squared_dict["clear"] = r_squared_clear
864
865 # Calculate kd and r squared for red
866 m_red, b_red = best_fit_slope_and_intercept(depths_list, red_log_list)
867 kd_red = m_red*(-1)
868 regression_line_red = [(m_red*x)+b_red for x in depths_list]
869 r_squared_red = coefficient_of_determination(red_log_list, regression_line_red)
870 kd_dict["red"] = kd_red
871 r_squared_dict["red"] = r_squared_red
872
873 # Calculate kd and r squared for green
```

```
874 m_green, b_green = best_fit_slope_and_intercept(depths_list, green_log_list)
875 kd_green = m_green*(-1)
876 regression_line_green = [(m_green*x)+b_green for x in depths_list]
877 r_squared_green = coefficient_of_determination(green_log_list,
regression_line_green)
878 kd_dict["green"] = kd_green
879 r_squared_dict["green"] = r_squared_green
880
881 # Calculate kd and r squared for blue
882 m_blue, b_blue = best_fit_slope_and_intercept(depths_list, blue_log_list)
883 kd_blue = m_blue*(-1)
884 regression_line_blue = [(m_blue*x)+b_blue for x in depths_list]
885 r_squared_blue = coefficient_of_determination(blue_log_list, regression_line_blue)
886 kd_dict["blue"] = kd_blue
887 r_squared_dict["blue"] = r_squared_blue
888
889 return (kd_dict, r_squared_dict)
890
891
892 def send_data_Sigfox(data):
893 """
894 Function to send data to Sigfox backend.
895
896 Parameters
897 ----------
898 data: bytes
899 list of bytes to send
900
901 Returns
902 -------
903 bool
904 True if data has been sent to Sigfox backend. Otherwise returns
905 False
906 """
907 # init Sigfox for RCZ1 (Europe)
908 sigfox = Sigfox(mode=Sigfox.SIGFOX, rcz=Sigfox.RCZ1)
909
910 # create a Sigfox socket
911 s = socket.socket(socket.AF_SIGFOX, socket.SOCK_RAW)
912
913 # make the socket blocking
914 s.setblocking(True)
915
916 # configure it as uplink only
917 s.setsockopt(socket.SOL_SIGFOX, socket.SO_RX, False)
918
919 bytes_data = len(data)
920 data_sent = 0
921
922 # wait until the module has joined the network
923 start = time.ticks_ms()
924 while data_sent < bytes_data:
925 # send bytes
```

```
926 data_sent = s.send(data)
927
928 # counter to send data to Sigfox. time_searching_Sigfox in seconds
929 if time.ticks_diff(start, time.ticks_ms()) > (time_searching_Sigfox * 1000):
930 print("possible timeout")
931 return False
932
933 return True
934
935
936 def read_request():
937 """ Function to send request to master
938 """
939 #p_en.value(0)
940
941 request = uart.read(1)
942 print("read request encoded={}".format(request))
943 return request
944
945
946 def send_request(slave_id=None):
947 """ Function to send request to a slave
948
949 Parameters
950 ----------
951 slave_id: char
952 name of the slave id
953 """
954 p_en.value(1)
955 time.sleep_ms(1000)
956
957 data = struct.pack('B', slave_id)
958 uart.write(data)
959
960 time.sleep_ms(100)
961 p_en.value(0)
962 print("request sent decoded= {}".format(data))
963 timing=uart.wait_tx_done(2000)
964
965
966 def read_data(slave_id=None):
967 """ Function to read data from a slave
968
969 Parameters
970 ----------
971 slave_id: char
972 name of the slave id
973 """
974 # List of Red, Green, Blue and Clear values,log values and temperature
975 # values from each slave
976
977 data = uart.read(36)
978
```

```
979 return data
980
981
982 def wait_requests():
983 """ Function to wait until something is received in the Rx pin
984 """
985 number_slaves=0
986 start = time.ticks_ms()
987
988 while uart.any() >= 0:
989 blink_led(1,500,led_orange)
990 request_received_encoded = read_request()
991 time.sleep_ms(100)
992 if request_received_encoded:
993 start = time.ticks_ms()
994 request_received = struct.unpack('B',request_received_encoded)
995 print("read decoded= {}".format(request_received))
996 number_slaves=number_slaves+1
997 if time.ticks_diff(start, time.ticks_ms()) > 10000:
998 return number_slaves
999
1000
1001 def wait_transmision():
1002 """ Function to wait until something is received in the Rx pin
1003 """
1004 start = time.ticks_ms()
1005 data_received = None
1006 while uart.any() >= 0:
1007 blink_led(1,500,led_orange)
1008 data_received_encoded = read_data()
1009 time.sleep_ms(100)
1010 if data_received_encoded:
1011 data_received = struct.unpack('BHfffffffff',data_received_encoded)
1012 print("read decoded= {}".format(data_received))
1013 return data_received
1014 if time.ticks_diff(start, time.ticks_ms()) > 20000:
1015 return data_received
1016
1017
1018 def collect_data(data=data_received):
1019 """ Function to collect the data received in a proper manner
1020 """
1021 slave_addr = data_received[0]
1022 temperature_value = data_received[1]
1023 red_value = data_received[2]
1024 green_value = data_received[3]
1025 blue_value = data_received[4]
1026 clear_value = data_received[5]
1027 log_red_value = data_received[6]
1028 log_green_value = data_received[7]
1029 log_blue_value = data_received[8]
1030 log_clear_value = data_received[9]
1031
```

```
1032 for i in range(len(slave_id_list)):
1033 if slave_addr==slave_id_list[i]:
1034 temperature_sensors_list[i]=temperature_value
1035 light_sensors_list[i][0]=red_value
1036 light_sensors_list[i][1]=blue_value
1037 light_sensors_list[i][2]=green_value
1038 light_sensors_list[i][3]=clear_value
1039 light_sensors_log_list[i][0]=log_red_value
1040 light_sensors_log_list[i][1]=log_blue_value
1041 light_sensors_log_list[i][2]=log_green_value
1042 light_sensors_log_list[i][3]=log_clear_value
1043
1044
1045 def variables_automatic():
1046 """ Change variables to fit with number of slaves
1047 """
1048 global slave_id_list, slave_id1, temperature_sensors_list, depth1, depths_list,
light_sensors_list, light_sensors_log_list, rgbc_list_sensor1
1049
1050 for i in range(number_slaves):
1051 slave_id_list.append(slave_id1)
1052 slave_id1=slave_id1+1
1053 temperature_sensors_list.append(0)
1054 depths_list.append(depth1)
1055 depth1=depth1+0.30
1056 light_sensors_list.append([0.0,0.0,0.0,0.0])
1057 light_sensors_log_list.append([0.0,0.0,0.0,0.0])
1058
1059 print("list of slaves: {}".format(slave_id_list))
1060 print("list of depths: {}".format(depths_list))
1061
1062
1063 def get_battery_status():
1064 """
1065 Function to get the battery voltage
1066
1067 Returns
1068 -------
1069 volt: int
1070 voltage from battery
1071 """
1072 i2c = machine.I2C(0, mode=I2C.MASTER, pins=('P22', 'P21'))
1073 py = Pytrack(i2c=i2c)
1074 volt = py.read_battery_voltage()
1075 i2c.deinit()
1076
1077 return volt
1078
1079
1080 def deep_sleep(time_to_deep_sleep):
1081 """
1082 Function to set Pytrack in ultra low power (deep sleep) during x time.
1083
```

```
Parameters
-------
time_to_deep_sleep: int
seconds to stay in deep sleep
"""
i2c = machine.I2C(0, mode=I2C.MASTER, pins=('P22', 'P21'))
py = Pytrack(i2c=i2c)
py.setup_sleep(time_to_deep_sleep)
py.go_to_sleep(gps=True)
i2c.deinit()

""" CODE """

# Start. LED green
blink_led(3, 500, led_green)
time.sleep_ms(100)

# Receive request from slaves and check the number of slaves
number_slaves = wait_requests()
time.sleep_ms(100)

variables_automatic()

# Send request to slaves
for j in range(len(slave_id_list)):
send_request(slave_id=slave_id_list[j])
time.sleep_ms(100)
blink_led(2,300,led_yellow)

# Read TCS34725 sensors and calculate kd and r-squared. LED pink
time.sleep_ms(1000)
for j in range(len(slave_id_list)):
data_received = wait_transmision()
collect_data(data=data_received)
time.sleep_ms(1000)
blink_led(2,300,led_blue)

print("Final temperature list: {}".format(temperature_sensors_list))
print("Final optic list: {}".format(light_sensors_list))
print("Final log_optic list: {}".format(light_sensors_log_list))
blink_led(1, 1000, led_pink)

kd_dict, r_squared_dict = kd_rsquared()
print("Kd and r_squared: {}, {}".format(kd_dict,r_squared_dict))
blink_led(1, 1000, led_orange)

# Get voltage and deployment sequence. LED pink


volt = get_battery_status()

deployment_seq = get_deployment_seq()
slaves = ord(get_slaves())
```

```
1137
1138 blink_led(1, 1000, led_blue)
1139
1140 # Get GPS and set datetime. LED yellow
1141 time.sleep_ms(1000)
1142 data_gps = get_lat_lon_datetime_gps(time_searching_GPS)
1143 set_datetime()
1144 blink_led(1, 1000, led_yellow)
1145
1146 # Send data to Sigfox. LED orange
1147 time.sleep_ms(1000)
1148 payload_gps = convert_data_to_payload_gps(device_id=device_id,
deployment=deployment_seq, lat=data_gps['latitude'],
1149 lon=data_gps['longitude'],
alt=data_gps['altitude'],
hdop=data_gps['hdop'])
1150 if precision_data:
1151 payload_data = convert_data_to_high_precision_payload(device_id=device_id,
deployment=deployment_seq,
1152 kd=kd_dict["clear"],
r_squared=r_squared_dict["cl
ear"],
1153 volt=volt)
1154 else:
1155 payload_data = convert_data_to_low_precision_payload(device_id=device_id,
deployment=deployment_seq,
1156 kd=kd_dict["clear"],
r_squared=r_squared_dict["cle
ar"],
1157 volt=volt)
1158
1159 payload_temperature_data = convert_temperature_data_to_payload(device_id=device_id,
deployment=deployment_seq,temperature=temperature_sensors_list)
1160
1161
1162 blink_led(1, 1000, led_orange)
1163
1164 # send GPS data only on this sequence deployments
1165
1166 if any(s in deployment_seq for s in ("a", "o")):
1167 gps_sent = send_data_Sigfox(bytes(payload_gps))
1168
1169 else:
1170 gps_sent = False
1171
1172
1173 data_sent = send_data_Sigfox(bytes(payload_data))
1174 temperature_data_sent = send_data_Sigfox(bytes(payload_temperature_data))
1175
1176
1177 """ time.sleep_ms(100) """
1178 blink_led(1, 1000, led_green)
1179
```

```
1180
1181 # Save data to SD. LED blue
1182 time.sleep_ms(1000)
1183 save_header()
1184 save_data(device_id=device_id, deployment=deployment_seq, datetime=rtc.now(),
lat=data_gps['latitude'],
1185 lon=data_gps['longitude'], alt=data_gps['altitude'],
hdop=data_gps['hdop'], volt=volt, data_sent=data_sent,
1186 temperature_data_sent=temperature_data_sent,
temperature_list=temperature_sensors_list, kd=kd_dict,
r_squared=r_squared_dict, raw_data=light_sensors_list)
1187
1188 blink_led(1, 1000, led_blue)
1189
1190 # Enter to deep sleep. LED red
1191 time.sleep_ms(1000)
1192 blink_led(1, 1000, led_red)
1193
1194 deep_sleep(time_to_deep_sleep)

1195
```

Anexo 2: Código Maestro

```
#Anexo 2: Código Esclavo

import machine
import micropython
import pycom
import time
import gc
from math import log
import socket
import os
import ubinascii
import struct
from pytrack import Pytrack
from machine import I2C, Pin, RTC, UART
from onewireb import DS18X20, OneWire

""" SET UP """

# Heartbeat LED flashes in blue colour once every 4s to signal that the system is alive
# Turn off firmware blinking
pycom.heartbeat(False)

# Time values in seconds of different variables
time_to_deep_sleep = 720

# TCS34725 integration time in ms: 2.4, 24, 50, 101, 154, 700
integration_time_TCS34725 = 154

# TCS34725 gain: 1, 4, 16, 60
gain_TCS34725 = 1

# Number of measures
number_measures = 10

# Boolean to calculate mean of light measures for each sensor
sensor_mean = True

# color leds
led_red = 0x7f0000
led_green = 0x007f00
led_yellow = 0x7f7f00
led_blue = 0x00FFFF
led_orange = 0xFF9900
led_pink = 0xFF00FF

# SDA and SCL pins
SDA_pin = 'P9'
SCL_pin = 'P21'

```

```
50 # Temperature Sensor pin
51 Temp_pin = 'P20'
52
53 # UART Comunication pins and initialization
54 p3 = Pin('P3', Pin.OUT)
55 p4 = Pin('P4', Pin.IN)
56 p_en = Pin('P8', mode=Pin.OUT)
57 p_en.value(0)
58
59 uart = UART(1, baudrate=9600)
60 uart.init(9600, bits=8, parity=None, stop=1)
61
62 # List of sensors values and list of log sensor values
63 sensors_list = []
64 sensors_log_list = []
65 temp_measured=0
66
67 # Slave ID
68 slave_addr = 97
69
70 # Value received from master
71 request = None
72
73 # Enable garbage collector
74 gc.enable()
75
76
77 """ FUNCTIONS """
78
79
80 def blink_led(times, ms, color):
81 """ Function to uses the RGB LED to make a blink light
82
83 Parameters
84 ----------
85 times: int
86 number of times to blink the light.
87 ms: int
88 time in ms that will be blink the light
89 color: string
90 type of color
91 """
92 for cycles in range(times):
93 pycom.rgbled(color)
94 time.sleep_ms(int(ms/2))
95 pycom.heartbeat(False)
96 time.sleep_ms(int(ms/2))
97
98
99 def clear_TCS34725_values():
100 """
101 Function to clear TCS34725 sensors list and log list
102
```

```
103 """
104 sensors_list.clear()
105 sensors_log_list.clear()
106
107
108 def clear_DS18X20_values():
109 """
110 Function to clear DS18X20 sensors values
111
112 """
113 temp_measured=0
114
115
116 def read_TCS34725_sensors():
117 """
118 Function to read light values from each sensor connected to I2C.
119 Create the sensors list with this values.
120
121 """
122 # to avoid errors in sipy, we need to import tcs34725 lib on this part of the code
123 import tcs34725
124
125 clear_TCS34725_values()
126
127 if sensor_mean:
128
129 mean_red = 0
130 mean_green = 0
131 mean_blue = 0
132 mean_clear = 0
133
134 # loop to do the number_measurements and calculate mean
135 for j in range(number_measures):
136 # I2C is a two-wire protocol for communicating between devices.
137 # At the physical level it consists of 2 wires: SCL and SDA, the clock
and data lines respectively.
138 # I2C objects are created attached to a specific bus.
139 # I2C initialized on bus 0
140 i2c = I2C(0, pins=(SDA_pin, SCL_pin))
141
142 try:
143 # Create a TCS34725 instance
144 sensor = tcs34725.TCS34725(i2c)
145 # set integration time as integration_time_TCS34725
146 sensor.integration_time(value=integration_time_TCS34725)
147 # and gain as gain_TCS34725
148 sensor.gain(gain_TCS34725)
149 red, green, blue, clear = sensor.read(raw=True)
150 # print("sensor {}: clear = {}".format(i+1, clear))
151
152 mean_red += red
153 mean_green += green
154 mean_blue += blue
```

```
155 mean_clear += clear
156
157 sensor.active(False)
158
159 except (OSError) as e:
160 print(e)
161
162 i2c.deinit()
163
164 try:
165 red = mean_red / number_measures
166 green = mean_green / number_measures
167 blue = mean_blue / number_measures
168 clear = mean_clear / number_measures
169 # print("sensor {}: mean clear = {}".format(i+1, clear))
170
171 log_red = log(red)
172 log_green = log(green)
173 log_blue = log(blue)
174 log_clear = log(clear)
175
176 except (ValueError) as e:
177 print(e)
178 red, green, blue, clear = tuple((0, 0, 0, 0))
179 log_red = 0
180 log_green = 0
181 log_blue = 0
182 log_clear = 0
183
184 # add values to list
185 sensors_list.extend([red, green, blue, clear])
186 sensors_log_list.extend([log_red, log_green, log_blue, log_clear])
187
188 else:
189 # I2C is a two-wire protocol for communicating between devices.
190 # At the physical level it consists of 2 wires: SCL and SDA, the clock and
data lines respectively.
191 # I2C objects are created attached to a specific bus.
192 # I2C initialized on bus 0
193 i2c = I2C(0, pins=(SDA_pin, SCL_pin))
194
195 try:
196 # Create a TCS34725 instance
197 sensor = tcs34725.TCS34725(i2c)
198 # set integration time as integration_time_TCS34725
199 sensor.integration_time(value=integration_time_TCS34725)
200 # and gain as gain_TCS34725
201 sensor.gain(gain_TCS34725)
202 red, green, blue, clear = sensor.read(raw=True)
203 print("sensor {}: clear = {}".format(i+1, clear))
204 log_red = log(red)
205 log_green = log(green)
206 log_blue = log(blue)
```

```
log_clear = log(clear)

sensor.active(False)

except (OSError, ValueError) as e:
print(e)
red, green, blue, clear = tuple((0, 0, 0, 0))
log_red = 0
log_green = 0
log_blue = 0
log_clear = 0

# add values to list
sensors_list.extend([red, green, blue, clear])
sensors_log_list.extend([log_red, log_green, log_blue, log_clear])

i2c.deinit()

print("light_sensor:")
print(sensors_list)
print(sensors_log_list)


def read_DS18X20_sensors():
"""Function to read the temperature with a DS18B20 sensor with onewire"""

global temp_measured
clear_DS18X20_values()

try:
while((temp_measured==8500)or(temp_measured==0)):
ow = OneWire(Pin(Temp_pin))
temp = DS18X20(ow)
temp_measured=temp.read_temp_async()
time.sleep(1)
temp.start_conversion()
time.sleep(1)
print("temperature_sensor: ")
print(temp_measured)
return temp_measured
except (OSError, ValueError) as e:
print(e)
temp_measured=0


def send_data(slave_id= None, temperature=None, sensors_list=None,
sensors_log_list=None):
""" Function to send data to master

Parameters
----------
data: int
values to send to master
```

```
259
260 """
261 p_en.value(1)
262 time.sleep_ms(100)
263
264 data = struct.pack('BHfffffffff', slave_id, temperature, sensors_list[0],
sensors_list[1], sensors_list[2], sensors_list[3]
265 , sensors_log_list[0], sensors_log_list[1],
sensors_log_list[2], sensors_log_list[3])
266 uart.write(data)
267
268 time.sleep_ms(100)
269 p_en.value(0)
270 print("data sent encoded={}".format(data))
271 data_decoded=struct.unpack('BHfffffffff',data)
272 print("data sent decoded={}".format(data_decoded))
273 timing=uart.wait_tx_done(2000)
274
275
276 def send_request(slave_id=None):
277 """ Function to send request to a slave
278
279 Parameters
280 ----------
281 slave_id: char
282 name of the slave id
283 """
284 p_en.value(1)
285 time.sleep_ms(100)
286
287 data = struct.pack('B', slave_id)
288 uart.write(data)
289
290 time.sleep_ms(100)
291 p_en.value(0)
292 print("request sent encoded= {}".format(data))
293 print("request sent decoded= {}".format(data))
294 timing=uart.wait_tx_done(2000)
295
296
297 def read_request():
298 """ Function to send request to master
299 """
300 #p_en.value(0)
301
302 request = uart.read(1)
303 print("read request encoded={}".format(request))
304 return request
305
306
307 def wait_transmision():
308
309 start = time.ticks_ms()
```

```
310
311 while uart.any() >= 0:
312 blink_led(1,500,led_orange)
313 request = read_request()
314 time.sleep_ms(100)
315
316 if request:
317 int_value = struct.unpack('B', request)[0]
318 print("read request decoded= {}".format(int_value))
319 break
320 if time.ticks_diff(start, time.ticks_ms()) > 1000:
321 return
322
323
324 def deep_sleep(time_to_deep_sleep):
325 """
326 Function to set Pytrack in ultra low power (deep sleep) during x time.
327
328 Parameters
329 -------
330 time_to_deep_sleep: int
331 seconds to stay in deep sleep
332 """
333 i2c = machine.I2C(0, mode=I2C.MASTER, pins=('P22', 'P21'))
334 py = Pytrack(i2c=i2c)
335 py.setup_sleep(time_to_deep_sleep)
336 py.go_to_sleep(gps=True)
337 i2c.deinit()
338
339
340 """ CODE """
341
342 # Start. LED green
343 blink_led(3, 500, led_green)
344
345 # To be sure that Master is reading
346 time.sleep_ms(3000)
347
348 # Send request to Master
349 send_request(slave_id=slave_addr)
350 time.sleep_ms(1000)
351
352 # Confirm request from Master
353 wait_transmision()
354 time.sleep_ms(1000)
355 blink_led(1,500,led_red)
356
357 # Read TCS34725 and DS18X20 sensors and calculate kd and r-squared. LED pink
358 read_TCS34725_sensors()
359 blink_led(1, 1000, led_pink)
360
361 time.sleep_ms(1000)
362 temp_measured=read_DS18X20_sensors()
```

```
363
364 # 10 seconds to asure that master is reading
365 time.sleep_ms(13000)
366 blink_led(1, 1000, led_yellow)
367
368 # Send Data through UART Communication
369 send_data(slave_id=slave_addr, temperature=temp_measured, sensors_list=sensors_list,
sensors_log_list=sensors_log_list)
370 time.sleep_ms(1200)
371 blink_led(3,300,led_orange)

372
```

Anexo 3: Código SigFox

```
#Anexo 3: Código SigFox


import requests
import datetime
import sqlite3
import math
from sqlite3 import Error


class Sigfox_to_db:
"""Class to create an SQL database from Sigfox data
It contains functions related to import data from Sigfox
and save it in a SQL database
"""

def __init__(self, url, devices):
"""It creates the instance of following variables:
url -- a string that contains API url
"""
self.url = url
self.devices = devices
self.json_list = []
self.counter = 0

@property
def jason_list(self):
return self.json_list

def sql_connection(self):
"""
Open connection to SQLite3 database
"""
try:
con = sqlite3.connect('mydatabase.db')
return con
except Error:
print(Error)
# Close db connection

def get_user_pwd_from_properties(self):
"""
Obtain user and password from properties.txt file
"""
f = open(r"sigfox_to_database\sigfox_to_database\properties.txt", "r")
user = f.readline().strip()
password = f.readline().strip()

return (user, password)

```

```
51 def get_and_list_jsons(self, url_in, login_in, password_in, pages):
52 """
53 Parameters
54 ----------
55 url_in: string
56 A string with the url Sigfox connection
57
58 login_in: string
59 A string that contains login Sigfox connection
60
61 password_in: string
62 A string that contains password Sigfox connection
63
64 pages: integer
65 Number of pages to check pagination
66
67 Returns
68 -------
69 boolean
70 Return False if there is a KeyError
71 """
72 self.counter += 1
73
74 try:
75 r = requests.get(url_in, auth=(login_in, password_in))
76 info = r.json()
77 self.json_list.append(info)
78
79 except requests.exceptions.RequestException as e:
80 print(e)
81 exit()
82
83 # iterate over all pages
84 try:
85 print("page number: {}".format(self.counter))
86 if self.counter == pages:
87 return True
88 # Recursive case
89 self.get_and_list_jsons(info['paging']['next'], login_in,
90 password_in, pages)
91 except KeyError:
92 return False
93
94 def get_data(self, messages_dict):
95 """Iterate over dictionary of a device's messages to get data
96
97 Parameters
98 ----------
99 messages_dict : dict
100 A dict with all the info we retrieve from device's messages
101
102 Returns
103 -------
```

```
104 messages_list: list
105 A list with dict that contains data in hexadecimal format and
106 timestamp in epoch format
107 """
108 messages_list = []
109 data = ''
110 time = ''
111 for key, value in messages_dict.items():
112 if key == 'data':
113 for index, value_dict_in in enumerate(value):
114 for k_in, v_in in value_dict_in.items():
115 if k_in == "data":
116 data = v_in
117 if k_in == "time":
118 time = v_in
119 if k_in == "device":
120 device = v_in
121 if k_in == "linkQuality":
122 lqi = v_in
123
124 messages_list.append([data, time, device, lqi])
125
126 return messages_list
127
128 def decoder(self, message):
129 """Function to decode the data message we got from Sigfox
130
131 Parameters
132 ----------
133 message : str
134 A string in hexadecimal format with data
135
136 Returns
137 -------
138 decoded_message: dict
139 A dict with decoded message
140 """
141 # declare an empty dictionary
142 decoded_message = {}
143 data = message[0]
144 time = message[1]
145 device = message[2]
146 lqi = message[3]
147
148 # ID as epoch time
149 identity = time
150
151 # Converting Epoch time into the datetime
152 time_utc_format = datetime.datetime.utcfromtimestamp(time).strftime(
153 '%Y-%m-%d %H:%M:%S')
154
155 # decoding to bytes
156 message_bytes = bytes.fromhex(message[0])
```

```
157
158 # access to data type
159 data_type = message_bytes[2]
160
161 try:
162 if data_type == 0:
163 # structure of payload:
164 # id(1) deploy(1) data_type(1) lat(3) lon(3) alt(2) hdop(1)
165 # --> 12 bytes
166 device_id = chr(message_bytes[0])
167
168 deployment = chr(message_bytes[1])
169
170 latitude_encoded = int.from_bytes(message_bytes[3:6], 'big')
171 latitude = ((latitude_encoded/16777215.0 * 180) - 90)
172
173 if latitude == -0.00000536441834242396:
174 latitude = 0
175
176 longitude_encoded = int.from_bytes(message_bytes[6:9], 'big')
177 longitude = ((longitude_encoded/16777215.0 * 360) - 180)
178
179 if longitude < 0:
180 longitude = 0
181
182 altitude = int.from_bytes(message_bytes[9:11], 'big', signed=True)
183
184 hdop = message_bytes[11]/10
185
186 decoded_message['id'] = identity
187 decoded_message['time'] = time_utc_format
188 decoded_message['device'] = device
189 decoded_message['lqi'] = lqi
190 decoded_message["device_id"] = device_id
191 decoded_message["deployment"] = deployment
192 decoded_message["data_type"] = data_type
193 decoded_message["latitude"] = latitude
194 decoded_message["longitude"] = longitude
195 decoded_message["altitude"] = altitude
196 decoded_message["hdop"] = hdop
197
198 if data_type == 1:
199 # structure of payload:
200 # id(1) deploy(1) data_type(1) kd(2) rsquared(2) volt(2) -->
201 # 9 bytes
202 device_id = chr(message_bytes[0])
203
204 deployment = chr(message_bytes[1])
205
206 kd_encoded = int.from_bytes(message_bytes[3:5], 'big')
207 kd = ((kd_encoded/10000 * 20) - 10)
208
209 rsquared_encoded = int.from_bytes(message_bytes[5:7], 'big')
```

```
210 r_squared = ((rsquared_encoded/10000 * 2) - 1)
211
212 voltage_encoded = int.from_bytes(message_bytes[7:9], 'big')
213 voltage = voltage_encoded / 10000 * 5
214
215 decoded_message['id'] = identity
216 decoded_message['time'] = time_utc_format
217 decoded_message['device'] = device
218 decoded_message['lqi'] = lqi
219 decoded_message["device_id"] = device_id
220 decoded_message["deployment"] = deployment
221 decoded_message["data_type"] = data_type
222 decoded_message["kd"] = kd
223 decoded_message["r_squared"] = r_squared
224 decoded_message["voltage"] = voltage
225
226 if data_type == 2:
227 # structure of payload:
228 # id(1) deploy(1) data_type(1) kd(3) rsquared(3) volt(3) -->
229 # 12 bytes
230 device_id = chr(message_bytes[0])
231
232 deployment = chr(message_bytes[1])
233
234 kd_encoded = int.from_bytes(message_bytes[3:6], 'big')
235 kd = ((kd_encoded/16777215.0 * 20) - 10)
236
237 rsquared_encoded = int.from_bytes(message_bytes[6:9], 'big')
238 r_squared = ((rsquared_encoded/16777215.0 * 2) - 1)
239
240 voltage_encoded = int.from_bytes(message_bytes[9:12], 'big')
241 voltage = voltage_encoded / 16777215.0 * 5
242
243 decoded_message['id'] = identity
244 decoded_message['time'] = time_utc_format
245 decoded_message['device'] = device
246 decoded_message['lqi'] = lqi
247 decoded_message["device_id"] = device_id
248 decoded_message["deployment"] = deployment
249 decoded_message["data_type"] = data_type
250 decoded_message["kd"] = kd
251 decoded_message["r_squared"] = r_squared
252 decoded_message["voltage"] = voltage
253
254 if data_type == 3:
255 # structure of payload:
256 # id(1) deploy(1) data_type(1) 6 temperatures (9 bytes) -->
257 # 12 bytes
258 device_id = chr(message_bytes[0])
259
260 deployment = chr(message_bytes[1])
261
262 temperature1=0
```

```
263 temperature2=0
264 temperature3=0
265 temperature4=0
266 temperature5=0
267 temperature6=0
268
269 if len(message_bytes)>3:
270 temperature1 = ((int(message_bytes[3])<<4)&0xFFF)
271 if len(message_bytes)>4:
272 temperature1 =
((int(message_bytes[3])<<4)&0xFFF)+((int(message_bytes[4])>>4)&0xF
)
273 temperature2 = ((int(message_bytes[4])<<8)&0xFFF)
274 if len(message_bytes)>5:
275 temperature2 =
((int(message_bytes[4])<<8)&0xFFF)+(int(message_bytes[5])&0xFF)
276 if len(message_bytes)>6:
277 temperature3 = ((int(message_bytes[6])<<4)&0xFFF)
278 if len(message_bytes)>7:
279 temperature3 =
((int(message_bytes[6])<<4)&0xFFF)+((int(message_bytes[7])>>4)&0xF
)
280 temperature4 = ((int(message_bytes[7])<<8)&0xFFF)
281 if len(message_bytes)>8:
282 temperature4 =
((int(message_bytes[7])<<8)&0xFFF)+(int(message_bytes[8])&0xFF)
283 if len(message_bytes)>9:
284 temperature5 = ((int(message_bytes[9])<<4)&0xFFF)
285 if len(message_bytes)>10:
286 temperature5 =
((int(message_bytes[9])<<4)&0xFFF)+((int(message_bytes[10])>>4)&0x
F)
287 temperature6 = ((int(message_bytes[10])<<8)&0xFFF)
288 if len(message_bytes)>11:
289 temperature6 =
((int(message_bytes[10])<<8)&0xFFF)+(int(message_bytes[11])&0xFF)
290
291
292 decoded_message['id'] = identity
293 decoded_message['time'] = time_utc_format
294 decoded_message['device'] = device
295 decoded_message['lqi'] = lqi
296 decoded_message["device_id"] = device_id
297 decoded_message["deployment"] = deployment
298 decoded_message["data_type"] = data_type
299 decoded_message["temperature1"] = temperature1/100
300 decoded_message["temperature2"] = temperature2/100
301 decoded_message["temperature3"] = temperature3/100
302 decoded_message["temperature4"] = temperature4/100
303 decoded_message["temperature5"] = temperature5/100
304 decoded_message["temperature6"] = temperature6/100
305
306
```

```
307
308 except KeyError:
309 return
310
311
312 return decoded_message
313
314 def insert_data_to_sqlite(self, message, con):
315 """Insert data to SQL database
316
317 Parameters
318 ----------
319 message: list
320 A list with data we want to insert in SQL database
321
322 Returns
323 -------
324 con: object
325 Object contains SQL connection
326 """
327 cursor = con.cursor()
328 is_data_type = False
329
330 try:
331 if 'data_type' in message:
332 is_data_type = True
333
334 except KeyError as err:
335 print(err)
336 is_data_type = False
337
338 if is_data_type:
339 if message['data_type'] == 0:
340 cursor.execute("CREATE TABLE IF NOT EXISTS gps(id integer PRIMARY
KEY,\
341 time text, device real, lqi text, device_id real,\
342 deployment text, data_type text, latitude real,\
343 longitude real, altitude real, hdop real)")
344
345 con.commit()
346
347 cursor.execute("insert or ignore into gps values (\
348 ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?\
349 )", [message['id'], message['time'], message['device'],
350 message['lqi'], message['device_id'],
351 message['deployment'], message['data_type'],
352 message['latitude'], message['longitude'],
353 message['altitude'], message['hdop']])
354
355 con.commit()
356
357 if message['data_type'] == 1:
358
```

```
359 cursor.execute("CREATE TABLE IF NOT EXISTS kd_low_precission(id
integer PRIMARY KEY,\
360 time text, device real, lqi text, device_id real,\
361 deployment text, data_type text, kd real,\
362 r_squared real, voltage real)")
363
364 con.commit()
365
366 cursor.execute("insert or ignore into kd_low_precission values (\
367 ?, ?, ?, ?, ?, ?, ?, ?, ?, ?\
368 )", [message['id'], message['time'], message['device'],
369 message['lqi'], message['device_id'],
370 message['deployment'], message['data_type'],
371 message['kd'], message['r_squared'],
372 message['voltage']])
373
374 con.commit()
375
376 if message['data_type'] == 2:
377
378 cursor.execute("CREATE TABLE IF NOT EXISTS kd_high_precission(id
integer PRIMARY KEY,\
379 time text, device real, lqi text, device_id real,\
380 deployment text, data_type text, kd real,\
381 r_squared real, voltage real)")
382
383 con.commit()
384
385 cursor.execute("insert or ignore into kd_high_precission values (\
386 ?, ?, ?, ?, ?, ?, ?, ?, ?, ?\
387 )", [message['id'], message['time'], message['device'],
388 message['lqi'], message['device_id'],
389 message['deployment'], message['data_type'],
390 message['kd'], message['r_squared'],
391 message['voltage']])
392
393 con.commit()
394
395 if message['data_type'] == 3:
396
397 cursor.execute("CREATE TABLE IF NOT EXISTS temperature(id integer
PRIMARY KEY,\
398 time text, device real, lqi text, device_id real,\
399 deployment text, data_type text, temperature1 real,\
400 temperature2 real, temperature3 real, temperature4 real,
temperature5 real, temperature6 real)")
401
402 con.commit()
403
404 cursor.execute("insert or ignore into temperature values (\
405 ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?\
406 )", [message['id'], message['time'], message['device'],
407 message['lqi'], message['device_id'],
```

```
408 message['deployment'], message['data_type'],
409 message['temperature1'], message['temperature2'],
410 message['temperature3'], message['temperature4'],
message['temperature5'], message['temperature6']])
411
412 con.commit()

413
```

Anexo 4: Código Inicialización SigFox

```
#Anexo 4: Código Inicialización SigFox


from sigfox_to_database import Sigfox_to_db


def main():

url = "https://backend.sigfox.com/api/"
pages = 1
device_id_list = [""]

sigfox2db = Sigfox_to_db(url, device_id_list)

# create connection to SQL database
con = sigfox2db.sql_connection()

# get Sigfox user and password
user, password = sigfox2db.get_user_pwd_from_properties()

# for each device in device_id_list
for device_id in device_id_list:

# set url
url = f"{sigfox2db.url}devices/{device_id}/messages"
sigfox2db.get_and_list_jsons(url, user, password, pages)

for json_data in sigfox2db.json_list:

messages_list = sigfox2db.get_data(json_data)

for message in messages_list:
decoded_message = sigfox2db.decoder(message)
#print(decoded_message)
sigfox2db.insert_data_to_sqlite(decoded_message, con)


if __name__ == "__main__":
main()

```